UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROYECTO FINAL DE CARRERA

GRADO EN INGENIERIA ELECTRÓNICA Y AUTOMÁTICA

EET, Terrassa, 20 de Diciembre de 2013

ARMONIZADOR PARA INSTRUMENTO MUSICAL

Estudiante:

Carlos Anchuela Arnalte

Tutores del proyecto:

Alber Masip Álvarez. Departamento de Control Automático

Manel Lamich Arocas. Departamento de Ingeniería Electrónica

A mis padres,

Angel y Angelines

Agradecimientos

Al profesor Albert Masip por ofrecerme la posibilidad de realizar este proyecto y por su apoyo, guía, colaboración y motivación durante la realización de este trabajo.

Al profesor Manel Lamich por darme la posibilidad de asistir a sus clases de DSPs y por su ayuda, consejos y dedicación durante la realización de este trabajo.

A todos los compañeros que han hecho que el camino hasta aquí haya sido menos complicado.

Y por supuesto, a mi familia y a mi novia por su compañía, cariño y consejos. Especialmente a mis padres por su interés y su apoyo incondicional en mis estudios.

Resumen

Este proyecto desarrolla una aplicación que permite generar una armonía a partir de una nota musical teniendo en cuenta la escala y la tonalidad.

El trabajo está dividido en tres partes. Primero se desarrolla la aplicación con el programa MATLAB para simular todo el proceso utilizando archivos de audio, después se implementa con un procesador digital de señales de Texas Instruments para trabajar en tiempo real utilizando una guitarra eléctrica y finalmente se diseña una interfaz gráfica con el programa Visual C++ que permite al usuario interactuar con la aplicación desde un PC.

Para trabajar en tiempo real se utiliza la técnica del doble buffer en la adquisición de datos de manera que se consigue reproducir la armonía a la vez que se toca el instrumento.

Para generar una armonía se debe conocer en todo momento la frecuencia fundamental de la nota que se toca. En este trabajo se utiliza la técnica del enventanado y la transformada de Fourier para determinarla. Esto se hace con una ventana que recorre la señal al mismo tiempo que se calcula la transformada de Fourier.

La elección del tamaño de la ventana junto con la frecuencia de muestreo juega un papel importante ya que determinan la resolución frecuencial, la resolución temporal y la frecuencia máxima que tiene la aplicación, o dicho de otra manera, determina el rango de notas del instrumento y la velocidad con que se pueden tocar.

Con la transformada de Fourier se obtiene el espectro de frecuencias de la señal donde aplicando un algoritmo se consigue detectar la frecuencia fundamental. A partir de esta frecuencia se determinan las frecuencias del resto de notas que forman un acorde o un intervalo armónico. Además, conociendo la amplitud de la señal cada vez que se determina la frecuencia fundamental se puede caracterizar la envolvente de la nota. Después, usando una señal de 1 Hz se generan todas las señales que forman la armonía donde se suman para obtener la señal de salida.

Entre los efectos que ofrece esta aplicación se encuentran los acordes de 10 escalas con 15 tonalidades por escala, intervalos de quinta, intervalos de quinta con octava, intervalos de octava, intervalos con dos octavas y la posibilidad de diseñar hasta tres intervalos armónicos simultáneos por nota a la elección del usuario.

Resum

Aquest projecte desenvolupa una aplicació que permet generar una harmonia a partir d'una nota musical tenint en compte l'escala i la tonalitat.

El treball està dividit en tres parts. Primer es desenvolupa l'aplicació amb el programa MATLAB per simular tot el procés utilitzant arxius d'àudio, després s'implementa amb un processador digital de senyals de Texas Instruments per treballar en temps real utilitzant una guitarra elèctrica i finalment es dissenya una interfície gràfica amb el programa visual C++ que permet a l'usuari interactuar amb l'aplicació des d'un PC .

Per treballar en temps real s'utilitza la tècnica del doble buffer en l'adquisició de dades de manera que s'aconsegueix reproduir l'harmonia alhora que es toca l'instrument.

Per generar una harmonia s'ha de conèixer en tot moment la freqüència fonamental de la nota que es toca. En aquest treball s'utilitza la tècnica del enfinestrat i la transformada de Fourier per determinar-la. Això es fa amb una finestra que recorre el senyal al mateix temps que es calcula la transformada de Fourier.

L'elecció de la mida de la finestra juntament amb la freqüència de mostreig juga un paper important ja que determinen la resolució freqüencial, la resolució temporal i la freqüència màxima que té l'aplicació, o dit d'una altra manera, determina el rang de notes de l'instrument i la velocitat amb què es poden tocar.

Amb la transformada de Fourier s'obté l'espectre de freqüències del senyal on aplicant un algoritme s'aconsegueix detectar la freqüència fonamental. A partir d'aquesta freqüència es determinen les freqüències de la resta de notes que formen un acord o un interval harmònic. A més, coneixent l'amplitud del senyal cada vegada que es determina la freqüència fonamental es pot caracteritzar l'envoltant de la nota. Després , usant un senyal d'1 Hz es generen tots els senyals que formen l'harmonia on es sumen per obtenir el senyal de sortida.

Entre els efectes que ofereix aquesta aplicació es troben els acords de 10 escales amb 15 tonalitats per escala, intervals de quintes, intervals de quintes amb octaves, intervals d'octava, intervals amb dues octaves i la possibilitat de dissenyar fins a tres intervals harmònics simultanis per nota a l' elecció de l'usuari .

Abstract

This project develops an application which creates a harmony from a musical note taking into account both scale and tonality.

The paper is divided into three parts. The first part accounts for the development of the application by means of the MATLAB software to simulate the whole process using audio files. The second accounts for the app implementation with a Texas Instruments' digital signal processor in order to work real time using an electric guitar. And the last part accounts for the design of a graphic interface by means of the program Visual C++, which allows the user to interact with the application from a PC.

In order to work real time, we used the double buffering method for data acquisition, so that a harmony is reproduced while the instrument is being played.

In order to generate a harmony, it is necessary to know the fundamental frequency in which the note is played. For this project, the windowing method and the Fourier Transform have been used to determine the note frequency. This is done by means of a window which runs through the signal while the Fourier Transform is calculated simultaneously.

The choice of the window size and the sampling frequency plays an important role in the process as they both determine the frequency resolution, the temporal resolution, and the maximum frequency of the application, in other words, they both determine the range of notes of the instrument and the speed they can be played on.

By means of the Fourier Transform we obtain the frequency spectrum of the signal, to which we can then apply an algorithm to detect the fundamental frequency. From this frequency, we can now determine the frequencies of the rest of notes which form a chord or a harmonic interval. Also, once we know the amplitude of the signal, we can then define the characteristics of the note envelope each time we determine the fundamental frequency. Then, using a 1 Hz signal, we can generate all the signals that form a harmony where they add to obtain the output signal.

Among the effects which this application offers, we can highlight the 10-scale chords with 15 tonalities in each scale, fifth intervals, fifth intervals on octave, octave intervals, two octave intervals, and the possibility of designing up to three simultaneous harmonic intervals per note according to user's choice.

Tabla de contenido

Listado de figuras

Listado de tablas

Listado de abreviaciones y acrónimos

ADC: Analog-to-Digital Converter
ADSR: Ataque Decaimiento Sostenimiento Relajamiento
CCS: Code Composer Studio
CD: Compact Disk
CPU: Central Processing Unit
DAC: Digital-to-Analog Converter
DFT: Discrete Fourier Transform
DSK: Digital Starter Kit
DSP: Digital Signal Procesor
EDMA: Enhanced Direct Memory Access
FFT: Fast Fourier Transform
FIR: Finite Impulse Response
Fs: Sampling frequency
HPS: Harmonic Product Spectral
JTAG: Joint Test Action Group
McBSP: Multichannel Buffered Serial Ports
MFC: Microsoft Foundation Class
RTDX: Real-Time Data Exchange
TI: Texas Instruments
VC++: Visual C++
WAV: WAVeform audio format

Unidades

dB: Decibelio
Hz: Hercio
MHz: Megahercio
ms: Milisegundo
Vp: Voltio de pico

Ecuaciones

A_0: Amplitud a la frecuencia fundamental
f_0: Frecuencia fundamental
f_{3a}: frecuencia de la tercera
f_{5a}: frecuencia de la quinta
f_{7a}: frecuencia de la séptima
f_{comp}: Frecuencia mínima de comparación
f_{Low}: Frecuencia de la nota en la octava más baja
f_{max}: Frecuencia máxima del instrumento
f_{min}: Frecuencia mínima del instrumento
f_{nota}: Frecuencia de la nota que se desea hallar
f_{Nota}: Frecuencia de una nota
frec. Mínima: Frecuencia mínima para reconstruir una señal
freferencia: Frecuencia de referencia La4 a 440 Hz
Fs: Sampling frequency
f_{tabla}: Frecuencía natural de la tabla para reconstruir la señal
h: Coeficientes del filtro FIR
i, k: índice o posición de un vector
L: Longitud de ventana
λ: Nivel de filtrado
long. Tabla: Longitud de la tabla para reconstruir la señal
n: Distancia en semitonos, entrada actual (para el filtro FIR)
N: Orden del filtro, número de muestras
nbits: número de bits
N_L: Número de muestras de la ventana
N_P: Número de muestra del paso
P: Paso de ventana
t: Tiempo
y: Frecuencia fundamental sin filtrar
$\hat{y}$: Frecuencia fundamental filtrada

Capítulo 1. Introducción

1.1 Acerca de la temática

Un armonizador para un instrumento musical es un efecto de audio que reproduce varias notas de forma simultánea creando una armonía a partir de una sola nota. La armonía se crea añadiendo más notas a la nota que se toca donde el tono de estas notas se genera a partir de la nota original teniendo en cuenta la escala y la tonalidad.

El hecho de cambiar el tono de una nota recibe el nombre de *Pitch Shifting*. Este efecto sube o baja el tono de la nota aumentando o disminuyendo su frecuencia. Existen dispositivos como el modulador de tono o *Pitch Shifter* que añaden varias notas a la original donde el tono de cada nota que se añade se determina a partir del tono original. Pueden hacer cambios de tono ya predefinidos, como por ejemplo subiendo o bajando una octava, o dar la opción a configurarlos, pero en ningún caso respeta la escala ni la tonalidad. El armonizador o *Harmonizer*, es un tipo de modulador de tono o *Pitch Shifter* pero además tiene en cuenta la escala y la tonalidad. Este hecho les otorga en ocasiones el nombre de armonizador o *Pitch Shifter* “inteligente”.

1.2 Antecedentes

El primer dispositivo capaz de generar una armonía disponible en el mercado surgió en 1975 y era el H910 Harmonizer de la casa *Eventide*. Se trataba básicamente de un modulador de tono o *Pitch Shifter* que ofrecía un cambio de tono de +/- una octava. Posteriormente, en 1986, *Eventide* fabricó el H3000 que fue el primer *Pitch Shifter* “inteligente”. Hay que decir que el término *Harmonizer* es el nombre que *Eventide* llamó a su *Pitch Shifter* "inteligente". Hoy en día *Eventide* sigue manteniendo sus derechos sobre la marca *Harmonizer* por eso el resto de fabricantes, como *Boss* o *Behringer* entre otros, denominan a su armonizador *Pitch Shifter* “inteligente”.

Gracias a la aparición de los procesadores digitales de señales o DSP se pudo mejorar el procesado de la señal de audio. Los primeros DSP se empezaron a utilizar en los ’80 pero no fue hasta los ‘90 cuando se empezaron a utilizar en el campo de los efectos de audio. *Eventide* fue una de las primeras empresas en fabricar procesadores

digitales de audio y empezó a desarrollar dispositivos y software de efectos con tecnología DSP en 1994. Desde entonces ha seguido desarrollando dispositivos de efectos como el DSP4000, el DSP7000 o el H7600 que incluyen el efecto armonizador.

1.3 Estado del arte

La forma de generar una armonía pasa por detectar la frecuencia fundamental de la nota que se toca. Existen varios métodos para hacerlo. Por un lado está el método de correlación en el dominio temporal que lo que hace es la autocorrelación de la señal para encontrar patrones repetitivos como la frecuencia fundamental pero tiene el inconveniente de que es sensible al ruido. Otra forma es analizando el espectro de frecuencias utilizando la transformada de Fourier ya que trabaja en el dominio frecuencial y puede asegurar una buena resolución frecuencial aunque a costa de perder resolución temporal. Por último también está la transformada wavelet que consigue un buen equilibrio entre la resolución frecuencial y temporal aunque este equilibrio se rompe en frecuencias muy bajas o muy altas.

1.4 Problemática

El principal problema que se plantea es encontrar un modo de detectar la frecuencia fundamental de las notas que se están tocando y debido a que la señal puede cambiar en el tiempo, se plantea otro problema y es que la aplicación debe trabajar en tiempo real sin perder eventualidades temporales.

Para determinar la frecuencia de la nota se propone utilizar la transformada de Fourier usando la técnica del enventanado, que aunque no es muy apropiada para tratar eventos en el tiempo, sí da buenos resultados a la hora de detectar la frecuencia y además es el método que se ha dado a conocer en la carrera. Por lo tanto, determinar la ventana que se va a utilizar será un punto clave para tener una buena resolución tanto frecuencial como temporal y no perder eventualidades temporales.

Para trabajar en tiempo real se implementará el armonizador con un procesador digital de señales o DSP de forma que se puedan adquirir y reproducir la señal al mismo tiempo que se procesan los datos.

1.5 Objetivos

1.5.1 General

Este proyecto tiene como objetivo desarrollar efectos de audio basados en la armonía musical. La aplicación debe detectar el tono de una nota de un instrumento musical y reproducir una armonía en función de la escala y la tonalidad.

1.5.2 Específicos

- Estudiar la formación de escalas y acordes.
- Estudiar la acústica de una nota musical.
- Simular el proceso con MATLAB antes del desarrollo con el DSP.
- Desarrollar la aplicación con el DSP DSK6713 de Texas Instruments.
- Utilizar la transformada de Fourier para determinar el espectro de la señal y desarrollar un algoritmo que permita detectar la frecuencia fundamental.
- Generar la armonía a partir de la frecuencia detectada.
- Formar acordes en función de la escala.
- Formar acordes en función de la tonalidad.
- Que la aplicación se pruebe y funcione al menos con un instrumento musical y se pueda utilizar en el rango de frecuencias del mismo.
- Que la aplicación interactúe con el instrumento en tiempo real.
- Desarrollar una interfaz gráfica con Visual C++ que permita al usuario interactuar con la aplicación en tiempo real.

1.6 Motivación

La motivación principal para desarrollar este trabajo es en parte por mi interés por la música. Un interés que había dejado aparcado hace mucho tiempo y que gracias a este proyecto vuelvo a retomar. Además la idea de combinar música y tecnología me resulta atractiva, y todavía más programando procesadores que quizás sea el área que más me guste de mi especialidad.

1.7 Herramientas de trabajo

1.7.1 Paca de desarrollo DSK6713

EL material hardware utilizado para realizar este proyecto es el DSK6713 de Texas Instruments (TI) ya que es el DSP que hay disponible en el laboratorio y con el que se imparten las clases.

Se utiliza un procesador DSP ya que, a diferencia de un procesador de propósito general, están especializados en procesar señales digitales en con una elevada potencia de cálculo realizando operaciones a gran velocidad, cosa que les hace apropiados para trabajar en tiempo real. Entre las aplicaciones digitales están las ventanas de adquisición, la transformada de Fourier, los filtros digitales, generación de formas de onda, etc. Todo esto les hace adecuados para tratar las señales y generar efectos de audio.

Esta placa permite desarrollar aplicaciones para la familia de DSP TI C67xx. Tal y como muestra la Figura 1.1 la placa de desarrollo esta divida en tres secciones.

- La sección del procesador, memoria y puertos de expansión. El procesador es el TMS320C6713 y está basado en la familia TMS320C6000. Este procesador está indicado para aplicaciones como el audio y puede opera a 225 MHz ejecutando hasta 1800 millones de instrucciones por segundo.
- La sección de alimentación y comunicaciones con el PC a través del puerto USB mediante el emulador JTAG.
- La sección de conversión analógico-digital y digital-analógica mediante el códec de audio AIC23. Éste códec tiene dos entradas y salidas analógicos.

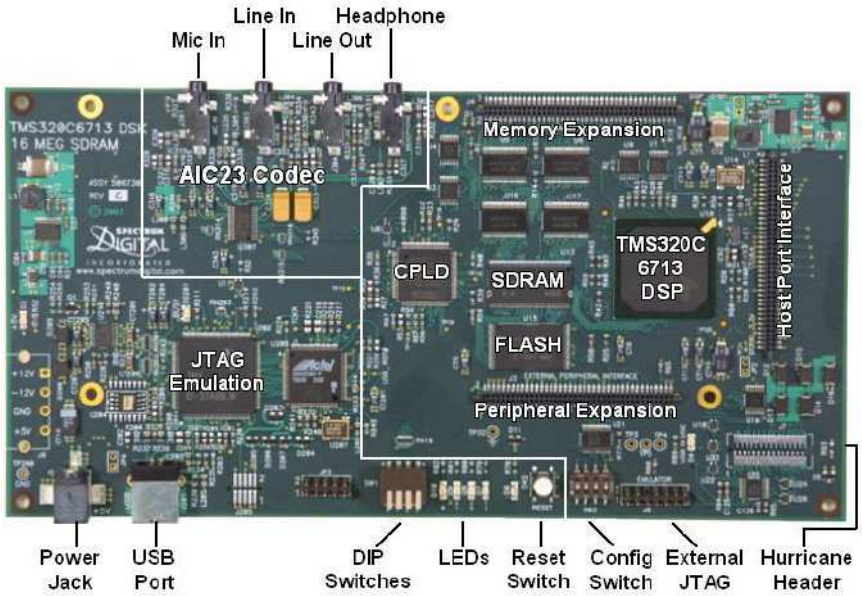


Figura 1.1 Placa de desarrollo DSK6713.

1.7.2 Software de desarrollo Code Composer Studio

El procesador se programa con el software Code Composer Studio (CCS). El CCS es el entorno de desarrollo para los DSP de Texas Instruments que permite trabajar con el lenguaje de programación ANSI C. Está diseñado para agilizar el proceso de desarrollo en aplicaciones en tiempo real. Incorpora herramientas de depuración y herramientas de emulación e intercambio de datos con aplicaciones externas y permite hacer gráficos del valor de las variables de programa. Permite optimizar el código en el momento de compilar el programa para pasarlo de código de alto nivel a código ensamblador. De esta forma se reduce el tiempo de procesado notablemente. Además incluye librerías con funciones ya implementadas para ayudar al programador.

1.7.3 Visual C++

Para que el usuario interactúe con la aplicación se decide realizar una interfaz gráfica en un PC ya que el DSP de Texas Instruments dispone de las librerías necesarias para poderse comunicar con Visual C++ (VC++).

VC++ es un software de Microsoft que permite desarrollar un entorno gráfico basado en diálogos (ventanas) donde el usuario interactúa con la aplicación a través de diferentes objetos como botones, barras de desplazamiento, casillas de verificación, cuadros de edición, etc. Es un lenguaje de alto nivel que incluye la biblioteca MFC (Microsoft Foundation Class) que es un conjunto de clases y funciones predefinidas e implementadas en código C++. En la programación, al momento que se van colocando los objetos en la ventana se puede crear automáticamente la función y la variable asociada a cada objeto.

Con el programa en marcha, éste queda a la espera de recibir un evento. Los eventos pueden ser, un click del ratón en un botón, arrastrar con el ratón un slider, hacer click con el ratón en una casilla de verificación, una entrada por teclado en un cuadro de edición, etc. Cuando se produce un evento se ejecuta sólo el código asociado a ese evento y que se encuentra dentro de la función asociada a ese objeto. Cuando finaliza el evento, el programa se queda de nuevo a la espera de recibir el siguiente evento.

1.7.4 MATLAB

Se ha escogido el software MATLAB de MatWorks para realizar la simulación porque es una herramienta potente de cálculo que además posee una librería o *toolbox* para el procesado digital de señales que será útil tanto en simulación como en el desarrollo de la aplicación DSP.

1.8 Estructura del proyecto

Este documento está formado por cinco capítulos y cinco apéndices. A continuación se resume el contenido principal de cada uno de ellos.

El capítulo uno introduce el proyecto, los objetivos y las herramientas de trabajo.

El capítulo dos introducir conceptos musicales para explicar cómo se hacen las escalas y como se forman los acordes y los intervalos armónicos. También explica las principales características acústicas de una nota musical.

En el capítulo tres se realiza todo el desarrollo de la aplicación. Éste está dividido en tres partes. En la primera parte se realiza la simulación de todo el proceso con MATLAB donde se desarrollan los algoritmos necesarios para reproducir un acorde. En la segunda parte se realiza la implementación con el DSK6713 teniendo en cuenta todo el trabajo realizado en simulación y adaptándolo al DSP para que trabaje en tiempo real, mejorando los algoritmos y creando las tablas para poder reproducir acordes según la escala y la tonalidad. En la tercera parte de diseña una interfaz gráfica con Visual C++ para seleccionar y configurar los efectos sin necesidad de parar la aplicación.

En el capítulo cuatro se hace un plan de trabajo sobre el tiempo que se ha dedicado a este proyecto y como se ha dividido el trabajo. También se hace un presupuesto de las horas de trabajo dedicadas a la programación de esta aplicación.

El capítulo cinco consiste en las conclusiones obtenidas durante el desarrollo del proyecto, se habla sobre las limitaciones de uso de la aplicación y se proponen una serie de mejoras que se pueden realizar en un trabajo futuro.

En el final del documento se encuentran cinco apéndices donde se dispone de una tabla con las frecuencias de las notas musicales hasta la séptima octava y los códigos que se ha utilizado para programar la simulación, la aplicación en tiempo real y la interfaz gráfica.

Capítulo 2. Marco teórico

Dado que este proyecto trata de desarrollar un armonizador para un instrumento musical, existen algunos conceptos teóricos relacionados con la música y la acústica. Antes de empezar con el desarrollo de la aplicación conviene introducir estos conceptos ya que son necesarios para un mejor entendimiento de este trabajo.

Primero se aborda el tema de la teoría musical. Ésta se basa en la utilizada en el mundo occidental ya que su uso está muy extendido en la mayoría de géneros musicales. Se definen conceptos importantes como los de nota, octava, semitono, tonalidad e intervalo. También se explica cómo se hacen las escalas y cómo se forman los acordes y los intervalos armónicos. Además se han generado dos tablas usando esta teoría, una tabla con todas las tonalidades por escala y otra con todos los intervalos para cada grado por escala para formar acordes.

Otro tema que se trata es el de la acústica musical. Básicamente se explican las características principales del sonido: tono, intensidad y timbre. También, entre otras cosas, se explica la relación que hay en frecuencia entre dos notas y la relación que hay entre la forma de una onda sonora y el teorema de Fourier.

2.1 Teoría musical

La música es el arte de ordenar y combinar los sonidos para transmitir sentimientos.

2.1.1 Notas musicales

Las notas musicales son doce sonidos. Siete notas son naturales y cinco son alteradas.

Las notas naturales son: *Do, Re, Mi, Fa, Sol, La, Si*.

Las notas alteradas son: *Do#/Reb, Re#/Mib, Fa#/Solb, Sol#/Lab, La#/Sib*. El símbolo # se denomina *sostenido* y hace la nota más aguda. El símbolo *b* se denomina *bemol* y hace la nota más grave. Por eso, hay notas que tienen el mismo sonido como *Do#* y *Reb*, ya que ambas se encuentran entre la nota *Do* y la nota *Re*.

Los doce sonidos colocados de forma ascendente, de lo grave a lo agudo, son:

Do - Do#/Reb - Re - Re#/Mib - Mi - Fa - Fa#/Solb - Sol - Sol#/Lab - La - La#/Sib - Si

Donde se puede observar que entre la nota *Mi* y la nota *Fa* no existen alteraciones. Aunque hay que decir que *Mi#* equivaldría a *Fa* y que *Fab* equivaldría a *Mi*.

Después de la doceava nota se vuelve a repetir toda la serie. Esto quiere decir que entre la nota *Si* y la nota *Do* tampoco hay alteraciones y por consiguiente, la nota *Si#* equivaldría a *Do* y la nota *Dob* equivaldría a *Si*.

Octava

Las notas naturales son una serie de siete notas donde la primera es *Do* y la séptima es *Si*. Después de la nota *Si* se vuelve a repetir la serie y así sucesivamente. Esto quiere decir que si el *Do* es la primera nota de la serie, el siguiente *Do* es la octava nota:

Do (1ª) - Re (2ª) - Mi (3ª) - Fa (4ª) - Sol (5ª) - La (6ª) - Si (7ª) - Do (8ª) - Re (9ª)…

Esto ocurre con cualquier nota de la serie. Por ejemplo si *Re* es la primera nota, el siguiente *Re* sería la octava. El termino octava se utiliza, entonces, para designar a una misma nota pero de una serie siguiente o de una serie anterior.

La octava también es aplicable a las alteraciones ya que la distancia es la misma:

Do# (1ª) - Re# (2ª) - Mi# (3ª) - Fa# (4ª) - Sol# (5ª) - La# (6ª) - Si# (7ª) - Do# (8ª)…

Dob (1ª) - Reb (2ª) - Mib (3ª) - Fab (4ª) - Sob# (5ª) - Lab (6ª) - Sib (7ª) - Dob (8ª)…

Para saber a qué octava pertenece una nota se pone un subíndice numérico a su derecha y toda la serie de doce notas, de *Do* a *Si*, lleva el mismo subíndice.

…Do_2 - $Do\#_2/Reb_2$ - Re_2 - $Re\#_2/Mib_2$… … Do_3 - $Do\#_3/Reb_3$ - Re_3 - $Re\#_3/Mib_3$…

Cuanto más alto sea este número, la nota será más aguda y cuanto más bajo sea este número la nota será más grave. Así, por ejemplo, Do_2 es más grave que Do_3 y Do_5 es más agudo que Do_3.

Las notas con la octava más baja tienen el subíndice 0 y las más altas tiene el subíndice 10.

2.1.2 Escalas

Una escala es un conjunto de notas ordenadas que se usa para componer una melodía y crear sus acordes.

Las escalas usadas en el mundo occidental son las escalas diatónicas y los modos gregorianos. Están formadas por siete notas más la repetición de la primera que sería la octava. A cada nota le corresponde un grado. Cada grado se identifica mediante un número romano, del I al VII, e indica la posición de la nota dentro de la escala.

Grados	I	II	III	IV	V	VI	VII	I
Notas	Re	Mi	Fa#	Sol	La	Si	Do#	Re

Tabla 2.1 Notas y grados de una escala.

Cada escala tiene una estructura que la caracteriza y que viene dada por la distancia entre sus grados. Esta distancia puede ser de un tono o de un semitono, donde un semitono es la distancia mínima entre dos notas, como por ejemplo de *Do* a *Reb* o de *Mi* a *Fa*, y un tono es igual a dos semitonos, como por ejemplo de *Do* a *Re* o de *Mi* a *Fa#*.

Básicamente se usan dos escalas diatónicas, la mayor natural y la menor natural. De la escala menor natural derivan dos escalas más, que también son bastante utilizadas y que son las escalas menor armónica y menor melódica.

En la Tabla 2.2 puede verse la estructura que siguen las escalas diatónicas y la distancia de tono (T) o semitono (S) que hay entre sus grados. También se puede observar que la escala mayor natural y la menor natural tienen el mismo patrón o que siguen la misma secuencia de tono y semitono pero empezando en un grado diferente. En cambio, las escalas menor armónica y menor melódica no siguen esta secuencia ya que son escalas artificiales que derivan de la escala menor natural y que han sufrido una alteración de la distancia entre alguno de sus grados. Por ejemplo, la escala menor armónica tiene un tono y medio entre el sexto y el séptimo grado.

Escalas diatónicas	Estructura						
	I-II	II-III	III-IV	IV-V	V-VI	VI-VII	VII-I
Mayor natural	T	T	S	T	T	T	S
Menor natural	T	S	T	T	S	T	T
Menor armónica	T	S	T	T	S	T ½	S
Menor melódica	T	S	T	T	T	T	S

Tabla 2.2 Estructura de las escalas diatónicas.

Los modos gregorianos fueron muy utilizados en la Edad Media y en el Renacimiento y hoy en día aún son usados por algún género musical. Son escalas construidas a partir de cada uno de los grados de la escala diatónica mayor natural.

En la Tabla 2.3 puede verse la estructura de los modos gregorianos y la relación que guardan con la escala mayor natural.

Modos gregorianos	Estructura						
	I-II	II-III	III-IV	IV-V	V-VI	VI-VII	VII-I
Jónico	T	T	S	T	T	T	S
Dórico	T	S	T	T	T	S	T
Frigio	S	T	T	T	S	T	T
Lidio	T	T	T	S	T	T	S
Mixolidio	T	T	S	T	T	S	T
Eólico	T	S	T	T	S	T	T
Locrio	S	T	T	S	T	T	T

Tabla 2.3 Estructura de los modos gregorianos.

El modo *jónico* empieza en el primer grado de la escala mayor natural y por lo tanto su estructura es la misma. El modo *dórico* empieza en el segundo grado y por lo tanto la estructura es la misma que la de la escala mayor natural pero desplazada una posición a la izquierda. Así sucesivamente hasta el modo *locrio* y como detalle se puede observar que el modo *eólico* y la escala menor natural son iguales.

2.1.3 Tonalidad

Como se ha visto anteriormente una escala está formada por siete notas más la repetición de la primera u octava, donde cada una tiene una posición en la escala indicada por su grado, y que, entre nota y nota existe una distancia que viene definida por la estructura de la escala.

Este conjunto de notas se definen como tonalidad, donde la primera nota, que correspondiente al grado I, es la nota principal y se denomina tónica. Se dice entonces, que esa escala está en la tonalidad de esa nota.

En base a la nota tónica, se puede construir una escala respetando su estructura y teniendo en cuenta que en una escala no pueden faltar ni haber notas repetidas, tengan o no alteraciones, y en el caso de haber alteraciones, éstas deben ser todas con sostenidos o con bemoles. Esto hace que no siempre se pueda construir una escala en base a una nota determinada.

Por ejemplo, la escala mayor natural en tonalidad de Re sería:

Re - Mi - Fa# - Sol - La - Si - Do# - Re

Donde, la nota del primer grado es la misma que marca la tonalidad, es decir, la nota *Re* y las notas del resto de los grados vienen impuestas por la estructura de la escala. En este caso al tratarse de la escala mayor natural, su estructura es T - T - S - T - T - T - S. Esto quiere decir lo siguiente:

a) Entre el grado I y el grado II debe haber un tono (o dos semitonos):
   T - T - S - T - T - T - S
   Re *- Re#/Mib -* ***Mi*** *- Fa - Fa#/Solb - Sol - Sol#/Lab - La - La#/Sib - Si - Do - Do#/Reb.*

b) Entre el grado II y el III debe haber un tono:
   T - **T** - S - T - T - T - S
   Re - Re#/Mib - ***Mi*** *- Fa -* ***Fa#****/Solb - Sol - Sol#/Lab - La - La#/Sib - Si - Do - Do#/Reb*

   Fa# y *Solb* suenan igual, pero para hacer esta escala se debe coger el *Fa#*, ya que, como se ha dicho anteriormente, no pueden faltar notas. Esto además indicará que esta escala no tendrá bemoles.

c) Entre el grado III y el IV debe haber un semitono:
   T - T - **S** - T - T - T - S
   Re - Re# - Mi - Fa - ***Fa#*** *-* ***Sol*** *- Sol# - La - La# - Si - Do - Do#*

d) Entre el grado IV y el V debe haber un tono:
   T - T - S - **T** - T - T - S
   Re - Re# - Mi - Fa - Fa# - ***Sol*** *- Sol# -* ***La*** *- La# - Si - Do - Do#*

e) Entre el grado V y el VI debe haber un tono:
   T - T - S - T - **T** - T - S
   Re - Re# - Mi - Fa - Fa# - Sol - Sol# - ***La*** *- La# -* ***Si*** *- Do - Do#*

f) Entre el grado VI y el VII debe haber un tono:
   T - T - S - T - T - **T** - S
   Re - Re# - Mi - Fa - Fa# - Sol - Sol# - La - La# - ***Si*** *- Do -* ***Do#***

g) Entre el grado VII y el siguiente, que vuelve a ser el primero, debe haber un semitono:
T - T - S - T - T - T - **S**
*Re - Re# - Mi - Fa - Fa# - Sol - Sol# - La - La# - Si - Do - **Do#** - **Re** - Re# - Mi…*

La Tabla 2.4 resume todas las tonalidades por escala que puede haber:

Escala o modo	Tonalidad
Mayor natural / Jónico	Dob, Do, Do#, Reb, Re, Mib, Mi, Fa, Fa#, Solb, Sol, Lab, La, Sib, Si
Menor natural / Eólico	Do, Do#, Re, Re#, Mib, Mi, Fa, Fa#, Sol, Sol#, Lab, La, La#, Sib, Si
Menor armónica	Do, Do#, Re, Re#, Mib, Mi, Fa, Fa#, Sol, Sol#, Lab, La, La#, Sib, Si
Menor melódica	Do, Do#, Re, Re#, Mib, Mi, Fa, Fa#, Sol, Sol#, Lab, La, La#, Sib, Si
Dórico	Do, Do#, Reb, Re, Re#, Mib, Mi, Fa, Fa#, Sol, Sol#, Lab, La, Sib, Si
Frigio	Do, Do#, Re, Re#, Mib, Mi, Mi#, Fa, Fa#, Sol, Sol#, La, La#, Sib, Si
Lidio	Dob, Do, Reb, Re, Mib, Mi, Fab, Fa, Fa#, Solb, Sol, Lab, La, Sib, Si
Mixolidio	Do, Do#, Reb, Re, Mib, Mi, Fa, Fa#, Solb, Sol, Sol#, Lab, La, Sib, Si
Locrio	Do, Do#, Re, Re#, Mi, Mi#, Fa, Fa#, Sol, Sol#, La, La#, Sib, Si, Si#

Tabla 2.4 Escalas y tonalidades.

2.1.4 Armonía

La armonía son dos o más notas tocadas simultáneamente. Si se tocan dos notas se habla de intervalo armónico y si se tocan tres o más notas se llama acorde.

Acordes

El acorde más sencillo es la tríada y está formado por tres de las notas de una escala. Después le sigue la cuatríada que está formada por cuatro notas de una escala. La primera nota del acorde es la nota de referencia y puede ser cualquier nota de la escala, la segunda nota del acorde es la *tercera* nota de la escala empezando a contar por la nota de referencia, la tercera nota del acorde es la *quinta* nota de la escala empezando a contar por la nota de referencia y, en el caso de las cuatríadas, la cuarta nota es la *séptima* nota empezando a contar por la nota de referencia.

A la distancia entre la nota de referencia y cualquier otra nota se le llama intervalo y se mide en semitonos.

Por ejemplo, en la escala mayor natural en tonalidad de *Re*, el acorde cuatríada de *Re* como nota de referencia (grado I) sería el siguiente:

Escala: ***Re* (1ª)** *- Mi (2ª) -* ***Fa# (3ª)*** *- Sol (4ª) -* ***La (5ª)*** *- Si (6ª) -* ***Do# (7ª)***
Acorde cuatríada: Re - Fa# - La - Do#

Y los intervalos de cada nota respecto la de referencia:

Re *- Re# - Mi - Fa -* ***Fa#*** *- Sol - Sol# -* ***La*** *- La# - Si - Do -* ***Do#***
1er intervalo: De *Re* a *Fa#* = 4 semitonos
2º intervalo: De *Re* a *La* = 7 semitonos
3er intervalo: De *Re* a *Do#* = 11 semitonos

Y si el acorde fuera de *Si* (grado VI) de la misma escala y tonalidad:

Escala: ***Re (3ª)*** *- Mi (4ª) -* ***Fa# (5ª)*** *- Sol (6ª) -* ***La (7ª)*** *-* ***Si (1ª)*** *- Do# (2ª)*
Acorde cuatríada: *Si - Re - Fa# - La*

Y los intervalos de cada nota respecto la de referencia:

Si *- Do - Do# -* ***Re*** *- Re# - Mi - Fa -* ***Fa#*** *- Sol - Sol# -* ***La*** *- La#*
1er intervalo: De *Si* a *Re* = 3 semitonos
2º intervalo: De *Si* a *Fa#* = 7 semitonos
3er intervalo: De *Si* a *La* = 10 semitonos

Los intervalos de un acorde dependen entonces de la estructura de la escala. Esto quiere decir, que los intervalos entre la nota de referencia y el resto de notas del acorde son siempre igual para cada grado de la escala independientemente de la tonalidad. Es decir, los intervalos para el acorde en el grado I en la tonalidad de *Do* también son 4, 7,11 y los intervalos del acorde en el grado VI en tonalidad de *Do* son 3, 7,10.

En la Tabla 2.5 se resumen todas las escalas y los intervalos respecto a la nota de referencia por cada grado para formar acordes cuatríadas. En el caso de querer formar una acorde tríada no sería necesario el último intervalo, que es el de valor mayor.

Escala o modo	**Grado**						
	I	**II**	**III**	**IV**	**V**	**VI**	**VII**
Mayor natural / Jónico	4,7,11	3,7,10	3,7,10	4,7,11	4,7,10	3,7,10	3,6,10
Menor natural / Eólico	3,7,10	3,6,10	4,7,11	3,7,10	3,7,10	4,7,11	4,7,10
Menor armónica	3,7,11	3,6,10	4,8,11	3,7,10	4,7,10	4,7,11	3,6,9
Menor melódica	3,7,11	3,7,10	4,8,11	4,7,10	4,7,10	3,6,10	3,6,10
Dórico	3,7,10	3,7,10	4,7,11	4,7,10	3,7,10	3,6,10	4,7,11
Frigio	3,7,10	4,7,11	4,7,10	3,7,10	3,6,10	4,7,11	3,7,10
Lidio	4,7,11	4,7,10	3,7,10	3,6,10	4,7,11	3,7,10	3,7,10
Mixolidio	4,7,10	3,7,10	3,6,10	4,7,11	3,7,10	3,7,10	4,7,11
Locrio	3,6,10	4,7,11	3,7,10	3,7,10	4,7,11	4,7,10	3,7,10

Tabla 2.5 Intervalos para formar acordes.

Como los intervalos para una misma escala dependen del grado y no de la tonalidad, puede haber acordes que sean diferentes a pesar de que parten de la misma nota. Esto es porque, en función de la tonalidad, a esa nota le corresponde un grado diferente dentro de la escala. Por ejemplo, en la escala mayor natural la nota *Re* está, en la tonalidad de *Re* en el grado I, en la tonalidad de *Do* en el grado II y en la tonalidad de *Mib* en el grado VII.

En la Tabla 2.6 se muestra un ejemplo de todos los acordes de *Re* para las tonalidades de la escala mayor natural que tienen esta nota. Se puede apreciar que no siempre es igual y que la nota *Re* está en un grado diferente por lo que se puede afirmar que un acorde de una nota determinada depende de la escala y tonalidad.

Tonalidad	Acorde de Re	Intervalos	Grado
Re	Re - Fa# - La	4,7,11	I
Do	Re - Fa- La	3,7,10	II
Sib	Re - Fa- La	3,7,10	III
La	Re - Fa# - La	4,7,11	IV
Sol	Re - Fa# - La	4,7,10	V
Fa	Re - Fa - La	3,7,10	VI
Mib	Re - Fa - Lab	3,6,10	VII

Tabla 2.6 Acordes de *Re* en la escala mayor natural.

Intervalos armónicos

Si un intervalo era la distancia entre dos notas, el intervalo armónico son estas dos notas tocadas a la vez. Existen varios intervalos armónicos de especial importancia. Aquí se van a comentar básicamente dos, el intervalo armónico de quinta perfecta, o simplemente quintas, y el intervalo armónico de octava.

Algunos géneros musicales usan las quintas en vez de los acordes para acompañar la melodía. En las quintas hay una distancia de siete semitonos, por lo que las quintas de *Re* estarían formadas por las notas *Re* y *La*.

En los intervalos armónicos de octava hay una distancia de doce semitonos por lo que un intervalo armónico de octava estaría formado por la nota *Re* y la misma nota *Re* pero una octava más alta, o lo que es lo mismo, un Re_3 y un Re_4, por ejemplo.

Con el intervalo armónico de octava se consigue enriquecer mucho más el sonido. Es por eso que en ocasiones a las quintas se le suma el intervalo de octava. Entonces, unas quintas de *Re* más su intervalo armónico de octava estaría formado por las notas *Re*, *La* y nuevamente Re pero en una octava más alta. Si por ejemplo, el *Re* está en la tercera octava, las notas serían Re_3, La_3 y Re_4. Otra forma de combinar los intervalos armónicos y que causan un sonido interesante es hacer dos intervalos armónicos de octava, como por ejemplo Re_2, Re_3 y Re_4.

2.2 Acústica musical

El sonido que produce un instrumento no es más que una vibración o las variaciones de presión en un medio como el aire. Estas variaciones se transmiten o propagan en forma de onda sonora. Las características más importantes de esta onda son el tono, la intensidad y el timbre

2.2.1 Tono

El tono indica como de grave o agudo es el sonido. Se mide en hercios (Hz) y expresa la frecuencia de vibración del sonido. A más vibraciones, la frecuencia es más alta, el tono es más alto y el sonido es más agudo. A menos vibraciones, la frecuencia es más baja, el tono es más bajo y el sonido es más grave.

Por lo tanto, una nota musical es un sonido a una determinada frecuencia. Para saber la frecuencia de una nota se utiliza como referencia la nota *La₄* que tiene establecida una frecuencia de 440 Hz. La ecuación que permite hallar el resto de frecuencias de cada nota es:

$$f_{nota} = f_{referencia} \cdot \left(\sqrt[12]{2}\right)^{n} \quad (2.1)$$

Donde:

f_{nota} es la frecuencia de la nota que se desea hallar.
$f_{referencia}$ es la frecuencia de referencia *La₄* a 440 Hz.
$\sqrt[12]{2} = 1{,}059463094$ es la relación de frecuencias entre dos notas consecutivas.
n es la distancia en semitonos entre la frecuencia de referencia y la frecuencia de la nota que se desea hallar.

Así pues, la siguiente nota después del *La₄* sería *La#₄* (o *Sib₄*) y la distancia entre ambas es de un semitono, con lo que n es 1 y la frecuencia de *La#₄* igual a 466,163 Hz.

Para conocer la frecuencia de la nota *Sol#₄* (o *Lab₄*), que es la nota anterior al *La₄*, la distancia también es de un semitono, n también vale 1, pero en este caso n es negativa (-1) resultando una frecuencia de *Sol#₄* de 415,304 Hz.

De esta forma ya se pueden calcular las frecuencias de todas las notas en todas las octavas. Además, conociendo ya la frecuencia de una nota *x* se puede calcular la frecuencia de cualquier nota, tomando como frecuencia de referencia la de la nota *x*. Por ejemplo, si se quiere conocer la frecuencia de la nota *Sol#₄* partiendo de la frecuencia de la nota *La#₄*, $f_{referencia}$ sería igual a 466,163 Hz, el valor de n sería de -2, y el resultado daría 415,304 Hz, igual que con la frecuencia de referencia de *La₄* con n igual a -1.

Si se desea conocer la frecuencia de una misma nota pero de una octava superior, el valor de n es 12 por lo que la ecuación 2.1 queda:

$$f_{nota} = f_{referencia} \cdot 2 \tag{2.2}$$

Y si se desea conocer la frecuencia de una misma nota pero de una octava inferior, como el valor de n es -12, la ecuación 2.1 queda:

$$f_{nota} = \frac{f_{referencia}}{2} \tag{2.3}$$

Con lo que una nota tiene el doble de frecuencia de su octava inferior o la mitad de frecuencia de su octava superior. Entonces, si la frecuencia de la nota La_4 es 440 Hz, La_5 tiene una frecuencia de 880 Hz, La_6 de 1760 Hz, La_7 de 3520 Hz, etc. y La_3 de 220 Hz, La_2 de 110 Hz, La_1 de 55 Hz y La_0 de 27,5 Hz.

Como el oído humano capta frecuencias desde 20 Hz hasta 20000 Hz aproximadamente, el rango de octavas va desde cero hasta diez, siendo la frecuencia de la nota Do_0 de 16,351 Hz y la de la nota Mi_{10} de 21096,163 Hz.

Rango de Frecuencia de los instrumentos

Por su construcción, los instrumentos reproducen las notas en un rango determinado de frecuencias. En la Tabla 2.7 se muestran algunos ejemplos de instrumentos y su correspondiente rango de frecuencias aproximado.

Instrumento	Rango de frecuencias (Hz)
Bajo	41 – 294
Cello	65 – 698
Flauta	262 – 2349
Guitarra acústica	82 – 988
Guitarra eléctrica	82 – 1319
Piano	28 – 4186
Tambor	100 – 200
Trombón	73 – 587
Trompeta	164 – 988
Viola	131 – 1175
Violín	196 – 3136

Tabla 2.7 Rango de frecuencias de algunos instrumentos.

2.2.2 Intensidad

La intensidad se refiere a la presión que ejerce el aire en vibración sobre los oídos y está relacionada con la amplitud de la onda sonora. A más presión, más grande es la amplitud de la onda y el sonido se oirá más alto o fuerte, y a menos presión, más baja es la amplitud de la onda y el sonido se oirá más bajo o débil, llegando a no oírse si la amplitud es cero.

La intensidad es por lo tanto el volumen del sonido y se mide en decibelios (dB). El oído humano, normalmente, no es capaz de distinguir una variación de 6 dB. El umbral de audición son 0 dB y corresponde al mínimo sonido audible. El umbral del dolor está entre 120 y 140 dB a partir de los cuales se pueden producir daños en el oído.

Rango dinámico de un dispositivo de audio

El rango dinámico o relación señal/ruido es la diferencia entre el nivel máximo que un dispositivo de audio es capaza de emitir y el nivel de ruido cuando no hay señal. Cuanto mayor sea esta diferencia, mejor calidad tendrá el sonido del dispositivo de audio.

En un dispositivo de audio digital de 16 bits este rango no es mayor de 96 dB.

2.2.3 Timbre

El timbre podría definirse como el “color” del sonido y es la característica que permite distinguir una misma nota producida por diferentes instrumentos. Esto quiere decir que dos instrumentos pueden estar tocando con igual tono e intensidad, pero en cambio, no suenan igual, como por ejemplo un violín y una guitarra.

El timbre, propio de cada instrumento, depende de la forma de la onda sonora y de su envolvente.

Forma de onda y el teorema de Fourier

La onda sonora correspondiente a una nota en particular tiene una forma determinada. Pero en realidad, esta onda sonora es el resultado de la suma de una onda fundamental con múltiples ondas de diferentes intensidades y frecuencias que reciben el nombre de armónicos (segundo armónico, tercer, armónico, etc.). De ahí que también se haga la comparativa del timbre con el color, ya que un color es la suma de varios colores.

Con el teorema de Fourier se puede realizar el estudio de una onda sonora pudiendo descomponerla y analizar la onda fundamental y cada armónico por separado.

Gracias al teorema de Fourier se sabe que la forma de la onda sonora es el resultado de la suma de todos los armónicos con la onda fundamental y que cada una

de estas ondas por separado tiene forma sinusoidal. Además cada onda tiene una frecuencia y amplitud diferente, donde la onda fundamental generalmente tiene mayor amplitud y menor frecuencia siendo esta frecuencia la frecuencia de la nota.

A la frecuencia de la onda fundamental se le llama frecuencia fundamental y el resto de frecuencias de los armónicos son múltiplos enteros de la ésta frecuencia. Así pues, si el sonido de una nota tiene una frecuencia fundamental *f*, el segundo armónico tiene frecuencia *2f*, el tercero *3f*, el cuarto *4f* y así sucesivamente. Por ejemplo, la nota La_4 tiene como frecuencia fundamental 440 Hz, el segundo armónico 880 Hz, el tercer armónico 1320 Hz, el cuarto 1760 Hz, etc.

Entonces, la forma de onda de la onda sonora viene definida por los armónicos. En la Figura 2.1 se puede ver como se forma una onda constituida por una onda fundamental y 2 armónicos. En la Figura 2.1 A está representada la onda fundamental con una amplitud y frecuencia determinada. La Figura 2.1 B corresponde al segundo armónico con la mitad de amplitud y el doble de frecuencia de la onda A. En la Figura 2.1 C muestra el resultado de sumar las ondas A y B. La Figura 2.1 D corresponde al tercer armónico con mitad amplitud y el triple de frecuencia de la onda A. Finalmente, en la Figura 2.1 E es el resultado de sumarle a la onda C la onda D, obteniendo la forma de la onda final para este caso particular.

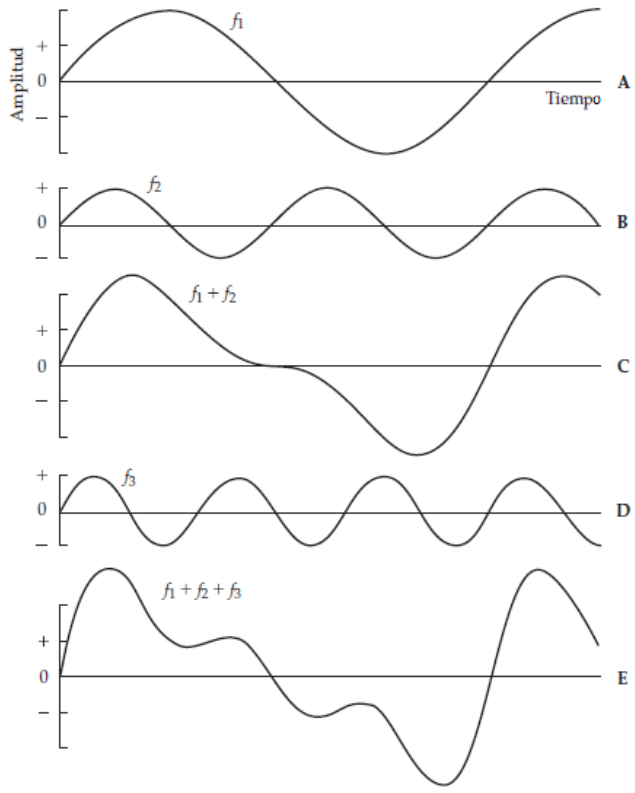


Figura 2.1 Combinación de ondas sinusoidales. (A) Frecuencia fundamental f_1. (B) Segundo armónico de frecuencia $f2 = 2f_1$ *y mitad amplitud de* f_1. (C) Suma de *f1* y *f2*. (D) Tercer armónico de frecuencia $f_3 = 3f_1$ y mitad amplitud de f_1. (E) Forma de onda resultando de la suma de f_1, f_2 y f_3.

Dependiendo del instrumento, puede haber más o menos armónicos y de amplitudes diferentes con lo que para cada instrumento se tiene una forma de onda diferente.

Normalmente un instrumento tiene bastantes más de tres armónicos pero hay algunos como el diapasón y dispositivos electrónicos capaces de generar una onda sonora formada únicamente por una onda fundamental. A esta onda se le llama *tono puro* y tiene una forma sinusoidal de una frecuencia y amplitud determinada.

Una forma de medir el timbre de un instrumento es mediante el espectro de frecuencias de la onda sonora. En el espectro se representa la amplitud y la frecuencia de cada onda por separado. En la Figura 2.2 Se puede ver el espectro de dos ondas sonoras. La Figura 2.2 A es un tono puro de frecuencia f_1. En cambio, en la Figura 2.2 B está representado el espectro de una onda sonora formada por la frecuencia fundamental f_1 y seis armónicos de frecuencias f_2 a f_7.

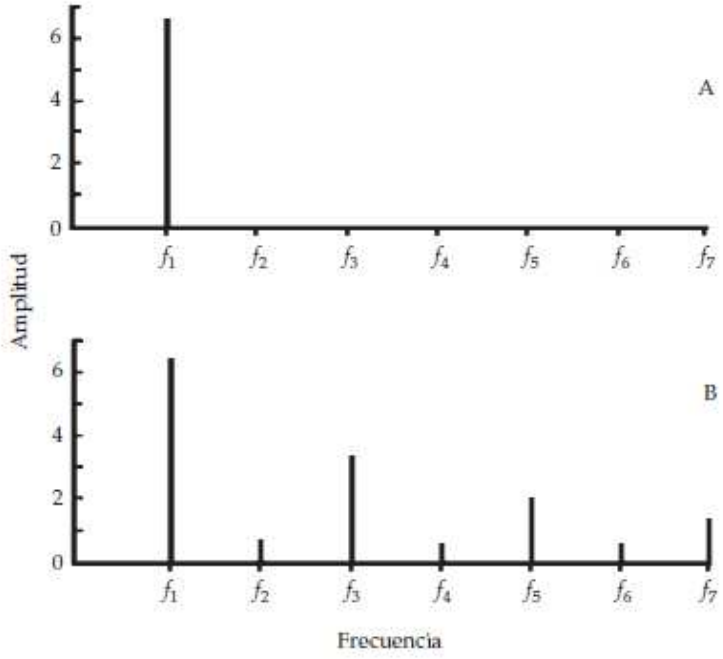


Figura 2.2 Espectro de ondas sonoras. (A) Tono puro.
(B) Onda compuesta.

Envolvente

La amplitud de la onda sonora no es siempre constante, puede variar a lo largo del tiempo. Es decir, en el momento de tocar una nota su amplitud al principio es alta, luego baja para mantenerse durante un tiempo y finalmente decae hasta que se apaga el sonido.

A esta variación de la amplitud se le llama envolvente y también es diferente en cada instrumento. Por ejemplo, un violín puede mantener durante más tiempo la nota que una guitarra.

Por lo tanto, la envolvente tiene diferentes fases en la línea temporal y que evolucionan en el siguiente orden:

a) El ataque (A): tiempo en que la onda llega a su máxima amplitud.
b) El decaimiento (D): tiempo en que la amplitud desciende a otro nivel.
c) El sostenimiento (S): tiempo que la amplitud se mantiene constante.
d) El relajamiento (R): tiempo en que la amplitud desciende hasta desvanecerse.

Debido al nombre de cada fase, a la envolvente también se le conoce con el nombre de ADSR. En la Figura 2.3 se puede ver como evoluciona la amplitud en el tiempo para cada fase del ADSR.

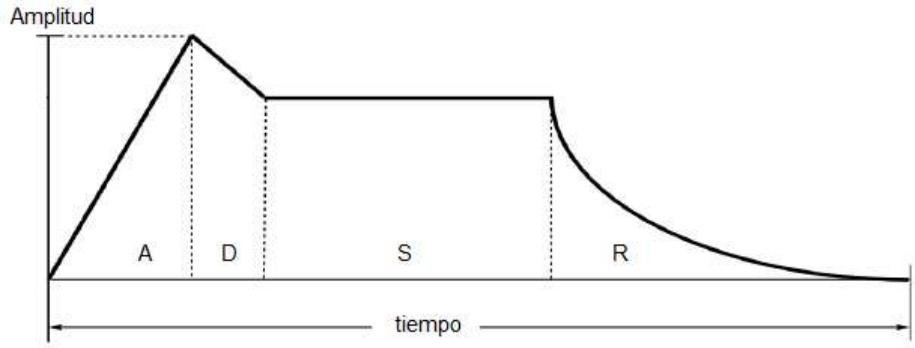


Figura 2.3 Envolvente o ADSR.

Capítulo 3. Desarrollo

En este proyecto se desarrollan varios efectos. Por un lado están los basados en el modulador de tono o *Pitch Shifter,* como quintas y octavadores para intervalos fijos, o el diseño de intervalos armónicos configurable por el usuario para intervalos variables. Por otro lado, están los efectos propios del armonizador, que reproducen acordes según la escala y la tonalidad teniendo en cuenta además, si las notas están fuera de la escala.

Aunque la aplicación puede funcionar con diversos instrumentos, las pruebas se realizan con una guitarra eléctrica. Pero hay que tener en cuenta que esta aplicación está implementada para que reproduzca tonos puros, por lo que si se utiliza otro instrumento, se puede comprobar que el resultado es el mismo.

La etapa de desarrollo se realiza en tres partes. Primero se hace una simulación con MATLAB para realizar un estudio previo y análisis de la aplicación. Después, se desarrolla la aplicación para que funcione en tiempo real con el DSP (Digital Signal Processor) de la placa de desarrollo DSK6713 programado con el software Code Composer Studio (CCS). En la última parte del desarrollo, se realizará una interfaz gráfica diseñada con Visual C++ que proporciona al usuario un entorno amigable desde un PC que le permite interactuar con la aplicación DSP en tiempo real gracias al protocolo RTDX.

En los apéndices se detalla toda la programación que se ha desarrollado para esta aplicación incluyendo comentarios sobre la configuración de registros del DSP. Para más detalle se sugiere consultar los *Datasheets* de Texas Instruments indicada en la bibliografía.

3.1 Simulación con MATLAB

El trabajo en simulación es el paso previo antes de empezar con la implementación de la aplicación con DSP. El objetivo es el de determinar los pasos a seguir y desarrollar los algoritmos necesarios para reproducir una señal de salida y generar una armonía simulando que se trabaja en tiempo real a partir de una señal de entrada.

En el transcurso de este apartado se tratan varios conceptos importantes como la transformada rápida de Fourier, el uso de ventanas, el paso de ventana, la resolución en frecuencia, el efecto ventana y su influencia, filtros discretos, etc. que serán necesarios para entender el trabajo en simulación y que servirán como base en el desarrollo de la aplicación con DSP.

El planteamiento pasa principalmente por las siguientes fases:

a) Adquisición de la señal de entrada.
b) Detección de la frecuencia fundamental y amplitud máxima.
c) Análisis de la frecuencia detectada.
d) Generar la señal de salida.

Primero se realiza la adquisición de datos con MATLAB para disponer de todas las muestras de la señal de entrada en un vector. Pero para simular que se trabaja en tiempo real se miran las muestras una a una y se utiliza una ventana que las recorre a intervalos de tiempo determinados para que no se pierdan eventualidades temporales. Dicha señal de entrada está formada por una nota o una secuencia de notas musicales guardadas en un fichero de audio de una duración determinada.

El siguiente paso es determinar la frecuencia fundamental y la amplitud de la señal de entrada para tener el tono y la envolvente de la nota, necesarios para generar la señal de salida. Para ello se utiliza el teorema de Fourier aplicado a señales en el tiempo discreto.

Después se analizan los resultados y se proponen diferentes alternativas a los problemas detectados.

Finalmente se estudia como generar la señal de salida y como determinar las frecuencias de las notas de un acorde a partir de la frecuencia fundamental en función de la escala y la tonalidad.

3.1.1 Adquisición de la señal de entrada

Para realizar las pruebas se han utilizado archivos de audio con las notas de un una guitarra generadas con el programa Guitar Pro.

Para tratar la señal de audio con MATLAB, estos archivos deben estar en formato WAV (WAVeform audio format). WAV es un formato digital sin compresión de datos el cual tiene una buena calidad de audio de CD y es adecuado para almacenar sonidos en un PC.

Al realizar la adquisición de datos con MATLAB la señal de entrada es muestreada pasando a tener valores discretos correspondientes a valores determinados de la amplitud de la señal de entrada.

También se obtienen otros parámetros de interés que pueden ser necesarios a la hora de tratar la señal como son la frecuencia de muestreo y el número de bits con que ha sido codificada. La frecuencia de muestreo o *Fs* (sampling frequency) suele ser de 44100 KHz para calidad de CD y el número de bits por muestra utilizados para codificar los datos en el archivo normalmente es de 16 bits.

Los datos adquiridos son guardados en un vector *entrada[k]* de longitud *N* muestras que depende de la *Fs* y de la duración *t* del archivo de audio. La ecuación que relaciona estos parámetros es:

$$Fs = \frac{N}{t} \tag{3.1}$$

Por lo que si el archivo tiene una duración *t* de 1 segundo y una *Fs* de 44100 KHz, la longitud del vector es de 44100 muestras.

3.1.2 Detección de la frecuencia fundamental y amplitud

La manera de determinar la frecuencia fundamental y su correspondiente amplitud es conociendo el espectro de frecuencias de la señal de entrada. Para ello se usará la transformada de Fourier, que es una herramienta matemática que permite pasar las señales del dominio temporal al dominio frecuencial.

Como la señal de entrada es muestreada, el método a utilizar es la transformada discreta de Fourier o DFT (Discrete Fourier Transform). La instrucción de MATLAB para determinar la DFT es la función *fft()* que aplica la transformada rápida de Fourier o FFT (Fast Fourier Transform) que es un algoritmo que calcula de una manera más eficiente y con menor coste computacional la DFT.

Enventanado

Con la transformada de Fourier no se puede saber la frecuencia en un instante de tiempo determinado, sólo se puede conocer el espectro de una señal dentro de un intervalo de tiempo dado, y cuanto más grande sea este intervalo (idealmente infinito), más información para hacer la FFT y mejor es el resultado. Además, este resultado es más fiable si la señal no cambia o es invariante a lo largo del tiempo, como por ejemplo mientras suena una misma nota.

Pero en la realidad ni se disponen de muestras infinitas ni las señales son estacionarias, ya que un músico puede cambiar de nota, prolongarla más o menos tiempo, hacer silencios, etc.

Debido a esto, no se puede aplicar la FFT a todo el vector *entrada[k]*, ya que en el archivo WAV puede haber más de una nota con lo que habría un solapamiento de frecuencias en el espectro. Sólo sería válido si en el archivo hubiera grabada una única nota pero en la realidad no será así, ya que, hay que tener en cuenta que se pretende trabajar en tiempo real y las notas pueden ir cambiando.

La solución pasa por realizar un enventanado, que es coger un número determinado de muestras de la señal correspondientes a un intervalo de tiempo suficiente como para que no coja dos notas. Por lo tanto, es necesaria una ventana de longitud *L* que vaya recorriendo la señal en el tiempo. La longitud de la ventana nos indicará el intervalo de tiempo de la señal que se coge y dependerá de la frecuencia mínima que se quiera detectar, de esta manera, se cogerá como mínimo un periodo de la señal.

$$L = \frac{1}{f_{min}} \tag{3.2}$$

Como se va a usar la guitarra como instrumento de pruebas, el valor de frecuencia mínima, correspondiente a la nota más baja posible en la guitarra, es un Mi_2 de 82,406889 Hz. Por lo tanto, aplicando la ecuación 3.2, la longitud de ventana *L* es igual a 0,012135 segundos.

Para saber el número de N_L muestras, se puede usar la ecuación 3.1 donde *L* es el tiempo *t* y *Fs* son 44100 Hz, por lo que la ventana es un vector *ventana[k]* que contiene unas 535 muestras aproximadamente. Esta es la cantidad de muestras que se utilizará para calcular la FFT.

Cada vez que se desplaza la ventana se realiza la FFT sólo de ese intervalo, es decir, de los valores que hay dentro de la ventana. Como resultado se obtiene una secuencia de espectros variables a lo largo del tiempo donde cada espectro está relacionado con una única nota, y que, mientras dure esa misma nota, los sucesivos espectros serán iguales en frecuencia. De esta manera se puede detectar la frecuencia fundamental de cada intervalo por lo que se pueden determinar las notas una a una.

Paso de ventana

El paso de ventana es el intervalo de tiempo que se deja pasar la señal, o dicho de otra forma, es el número de muestras que se desplaza la ventana para realizar la siguiente FFT. Es importante conocer cuánto valdrá el paso, ya que si es muy pequeño el procesador trabajará mucho y se podría saturar, y si es muy grande, podría perderse información.

Para determinar el paso se debe conocer la frecuencia máxima que se va a detectar y en el caso de la guitarra corresponde al Mi_6 de 1318,5100228 Hz. De esta manera no se deja pasar la señal más de un periodo.

$$P = \frac{1}{f_{max}} \tag{3.3}$$

Por lo tanto, aplicando la ecuación 3.3, el paso de ventana P es igual a 0,000758 segundos. Para saber el número de N_P muestras, se puede usar la ecuación 3.1 donde P es el tiempo t y Fs son 44100 Hz, por lo que el paso de ventana es de 33 muestras aproximadamente.

Esto quiere decir que se calcula la FFT cada 0,76 milisegundos aproximadamente y que se realizan unas 1315 FFT por segundo. Además, como la ventana es de 535 muestras de longitud y se desplaza cada 33 muestras hay un solapamiento de ventanas. Todo esto hace que sea difícil que se pierda información.

Resolución en frecuencia

La resolución en frecuencia depende de la frecuencia de muestreo y de la longitud de la ventana.

$$Resolución = \frac{Fs}{N_L} \tag{3.4}$$

Para el caso de la guitarra, con una Fs de 44100 Hz y una N_L de 535 muestras, aplicando la ecuación 3.4, la resolución es de 82,4 Hz aproximadamente. Esto implica un problema, porque la diferencia entre dos notas consecutivas es de unos 5 Hz a las frecuencias más bajas de la guitarra y de unos 74 Hz a las frecuencias más altas, con lo que se obtendría el mismo valor de frecuencia para diferentes notas.

Siendo Fs un parámetro fijo del archivo de audio, la solución pasaría por aumentar el tamaño de la venta. MATLAB permite modificar la longitud de la ventana en el momento de realizar la FFT de manera que se pueda tener una resolución de 1 Hz.

Esto puede implicar un mayor tiempo computacional, pero no supone un problema ya que no se pierde información porque los datos ya se encuentran en un vector.

Algoritmo de detección

Hay dos parámetros importantes que son necesarios a la hora de generar la señal de salida, estos son la frecuencia fundamental y su respectiva amplitud. Después de hacer la FFT, el resultado se guarda en un vector *FFT[k]* donde cada posición contiene el valor de la amplitud corresponde a una frecuencia. La frecuencia se puede determinar usando la ecuación 3.5.

$$frecuencia = i \cdot Resolución \tag{3.5}$$

Donde i es el índice correspondiente a una posición determinada del vector. Así, por ejemplo, para una resolución de 1 Hz, en la posición i = 4, la frecuencia es de 4 Hz y en esa posición del vector está el valor de amplitud correspondiente a esa frecuencia.

Hay que tener en cuenta que como se ha modificado la ventana, para tener una resolución de 1 Hz, el vector será de 44100 posiciones. Además, una particularidad de la FFT es que el resultado está en forma compleja y tiene simetría par. Esto quiere decir que se debe calcular el módulo y que la segunda mitad de los datos del vector son información redundante, por lo que la frecuencia máxima que se podrá detectar corresponde a la que hay en la mitad del vector y que viene expresada por la ecuación 3.6.

$$frecuencia\ máxima = \frac{N_L}{2} \cdot Resolución \tag{3.6}$$

O también, aplicando la ecuación 3.4 en la 3.6, la frecuencia máxima queda:

$$frecuencia\ máxima = \frac{Fs}{2} \tag{3.7}$$

Por lo que, en este caso, la frecuencia máxima que se puede detectar es de 22050 Hz y que corresponde a la posición del vector i = 22050.

El diagrama de flujo y el algoritmo que detecta la frecuencia fundamental son los de la Figura 3.1 y Figura 3.2 respectivamente. Este algoritmo recorre posición a posición el vector *FFT[k]* mirando el valor de la amplitud y cuando detecta el valor máximo, el índice del vector asociado a ese valor, corresponde con la frecuencia fundamental.

Como el instrumento de pruebas es una guitarra y la resolución es de 1 Hz, no es necesario que el algoritmo recorra las 22050 posiciones ya que para el rango de la guitarra, la primera posición sería la 82 y la última la 1319 aproximadamente, dependiendo de cómo esté afinado el instrumento, así que, dejando un margen de dos notas por delante y dos por detrás, con que se mire entre las posiciones 70 y 1500 ya es suficiente.

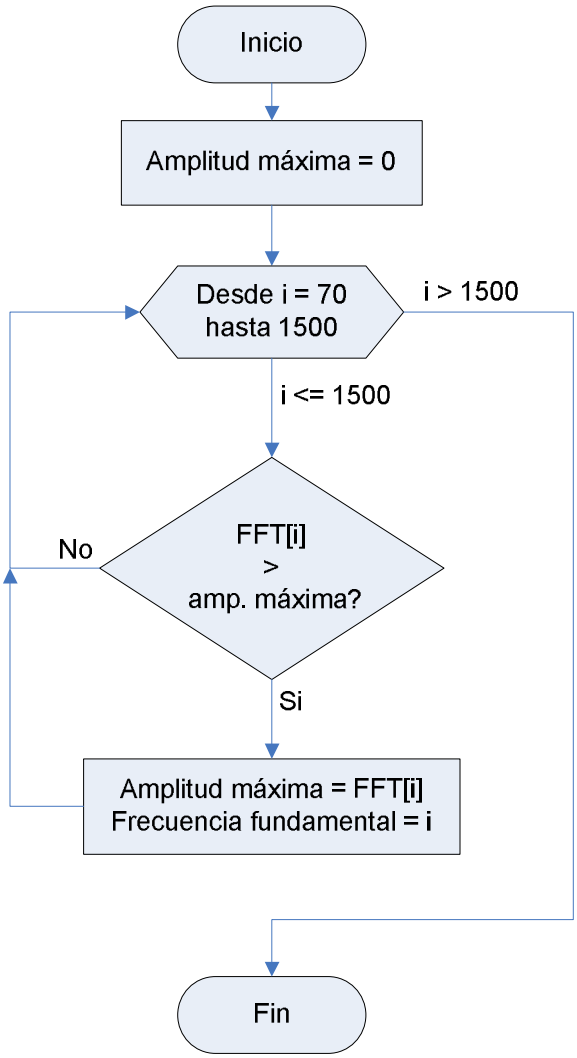


Figura 3.1 Diagrama de flujo del algoritmo de detección de la frecuencia fundamental y su amplitud.

```
amplitudMax = 0; % Se resetea la variable

for i=70:1500
    if FFT(i)>amplitudMax
        amplitudMax = FFT(i);
        frecFund = i;
    end
end
```

Figura 3.2 Algoritmo MATLAB de detección de la frecuencia fundamental y su amplitud.

Este algoritmo calcula la frecuencia fundamental cada vez que se llega al paso, es decir, cada 33 muestras o cada 0,76 milisegundos, por lo que existe un hueco de 33 muestras donde no se calcula la frecuencia fundamental, pero debido a que es un tiempo muy corto, se considerará que la frecuencia fundamental es igual a la calculada anteriormente.

3.1.3 Análisis de la frecuencia detectada

Silencios

Cuando hay un silencio, por ejemplo antes de empezar a tocar un instrumento o entre dos notas, a pesar de que la amplitud es cero aparecen picos de frecuencias o ésta empieza a oscilar debido a ruido. En la Figura 3.3 se muestran cómo evoluciona la amplitud y la frecuencia fundamental a lo largo del tiempo para la secuencia de notas La_3-La_3-Silencio-Re_4 y donde se puede observar que entre el La_3 y el Re_4 la frecuencia es irregular cuando la amplitud es cero.

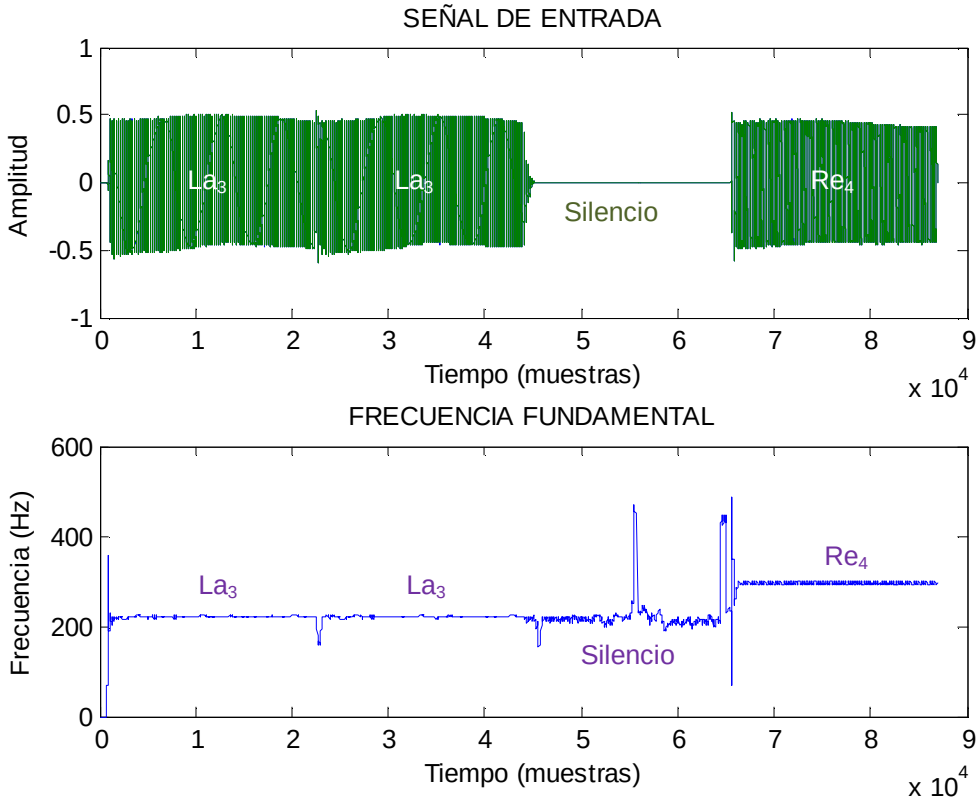


Figura 3.3 Amplitud y frecuencia fundamental para la secuencia de notas La_3-La_3-Silencio-Re_4.

Aunque es un efecto no deseado, en principio los silencios no deberían suponer ningún problema ya que a la hora de reproducir la señal de salida también se tiene en cuenta la amplitud y si ésta vale cero, la salida también valdrá cero. Pero para hacer más robusto el programa, se crea un algoritmo que lo que hace es ver el valor de todas las muestras de la señal de entrada que hay en la ventana, y si estas son cero, o están dentro de una tolerancia, no se calcula la FFT y directamente el valor de la frecuencia fundamental se pone a cero. Como en la adquisición de datos MATLAB da valores entre -1 y 1 a la señal de entrada, se considera un valor próximo a cero el de ± 0,05. En la Figura 3.4 se puede ver cómo queda la frecuencia después de aplicar el algoritmo.

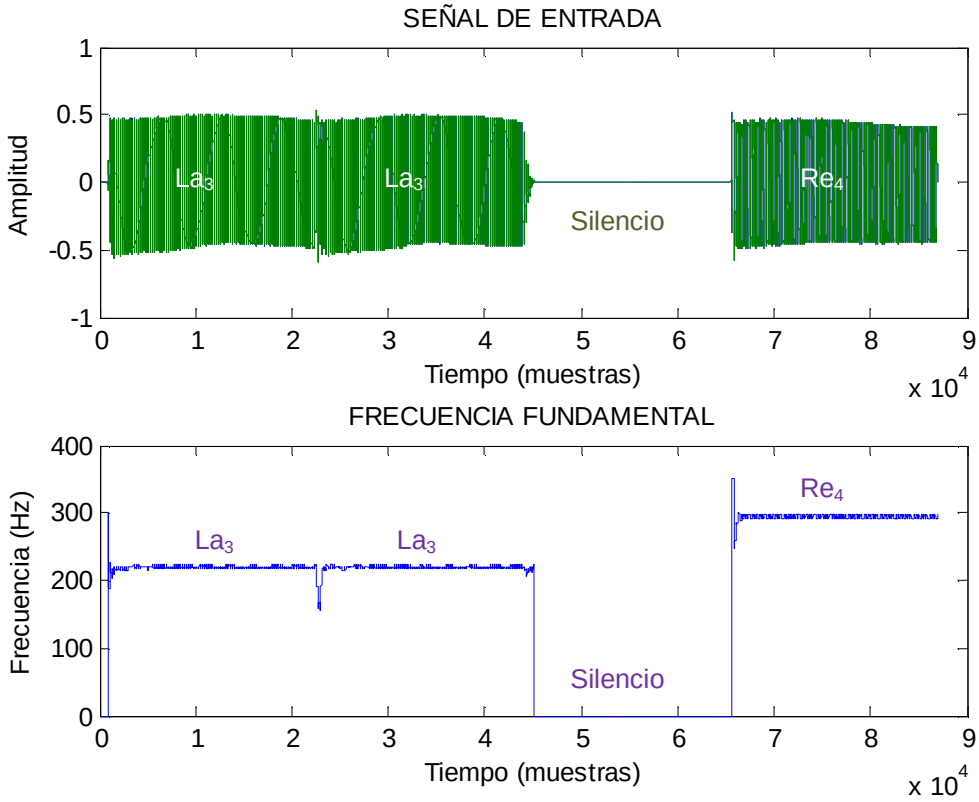


Figura 3.4 Amplitud y frecuencia fundamental para la secuencia de notas La_3-La_3-Silencio-Re_4 después de aplicar el algoritmo.

En la Figura 3.5 se muestra el diagrama de flujo y en la Figura 3.6 el algoritmo comentado anteriormente.

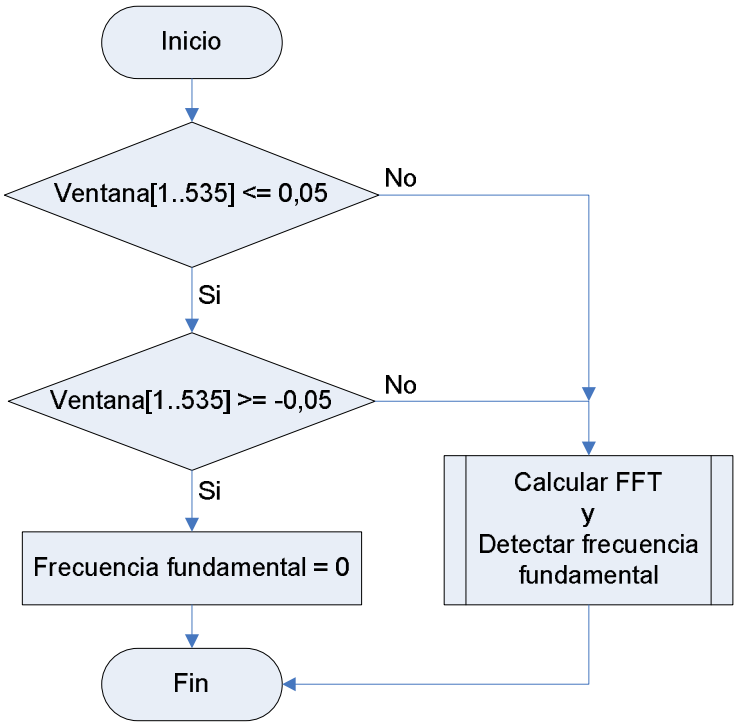


Figura 3.5 Diagrama de flujo del algoritmo de detección de silencios.

```
if ventana<=0.05 & ventana>=-0.05
    frecFund = 0;
else
% En caso contrario, se calcula la FFT y se aplica
% el algoritmo de detección.
    …
    …
end
```

Figura 3.6 Algoritmo MATLAB de detección de silencios.

La ventaja es que si la amplitud es cero, o próxima a cero, ya no es necesario calcular la FFT ni realizar el algoritmo de detección de la frecuencia fundamental, pero el inconveniente es que el hecho de comparar todos los valores de la ventana y que todos estén dentro de una tolerancia, implica mucho tiempo computacional.

Efecto ventana

En la Figura 3.7 se muestra cómo evoluciona la frecuencia fundamental a lo largo del tiempo para una secuencia de cuatro notas de guitarra de dos segundos de duración.

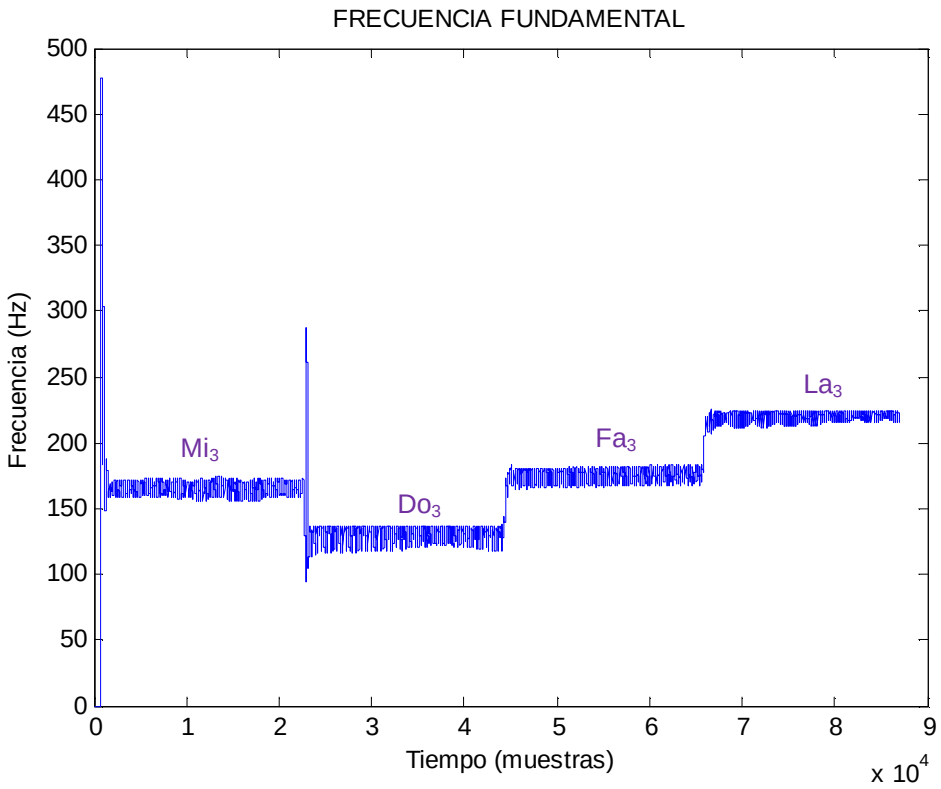


Figura 3.7 Frecuencia fundamental para la secuencia de notas Mi_3-Do_3-Fa_3-La_3.

Como se puede observar aparecen cuatro tramos donde cada uno corresponde a una nota de la secuencia. Las notas y los valores teóricos de frecuencia corresponden, para el primer tramo a un Mi_3 de 164,8 Hz, para el segundo a un Do_3 de 130,8 Hz, para el tercero a un Fa_3 de 174,6 Hz y para el cuarto a un La_3 de 220 Hz donde, a priori, cada uno de estos valores corresponde con cada tramo de la gráfica.

Sin embargo, haciendo un zoom en cualquier tramo de la Figura 3.7 se puede apreciar que la frecuencia oscila en torno al valor teórico. Esto puede verse en la Figura 3.8 donde se ha hecho un zoom de la mitad del primer tramo correspondiente a la nota Mi_3 de 164,8 Hz.

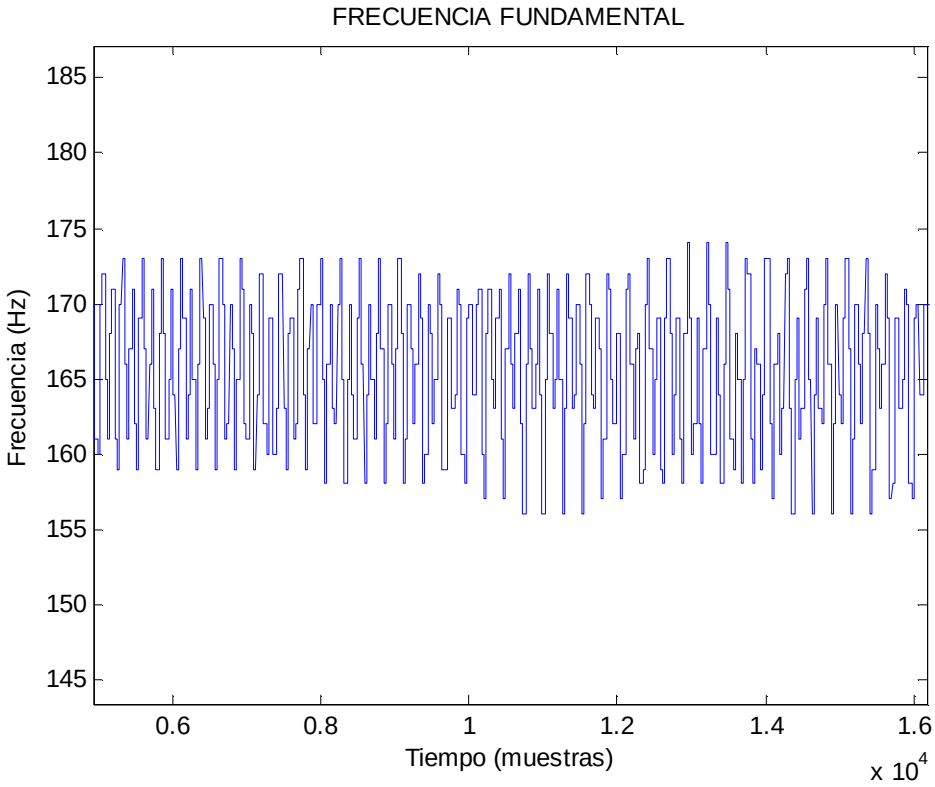


Figura 3.8 Zoom de la frecuencia fundamental correspondiente a la nota Mi_3 de 164,8 Hz.

En este caso la frecuencia tiene una variación 18 Hz aproximadamente que corresponderían a una o dos notas de diferencia por arriba y por abajo entorno al valor teórico. Esto quiere decir que estarían sonando varias notas muy cercanas entre sí en torno a la nota central en períodos muy cortos de tiempo con lo que la sensación sería la de escuchar la nota teórica con ruido.

Esto es porque cada vez que se hace la FFT el resultado no es siempre exactamente el mismo debido principalmente al efecto ventana. Para solucionar este problema se propone, por un lado cambiar el tipo de ventana, por otro filtrar la frecuencia fundamental y finalmente establecer la nota con un valor fijo de frecuencia.

Tipo de ventana

El efecto ventana es producido principalmente por el tipo o forma de ventana empleada donde también hay que considerar el tamaño de ésta. El hecho de cortar bruscamente el inicio y el final del intervalo que se pretende analizar equivale a considerar otra forma de onda distinta ya que lo que se mide no es la señal de entrada sino el producto de la misma por la ventana. Una multiplicación en el dominio temporal

(que es lo que se hace al enventanar) supone una convolución en el dominio frecuencial, es decir, el espectro que resulta es la convolución del espectro de la señal de entrada y el espectro de la ventana por lo que las características frecuenciales de la señal quedan alteradas por dicha ventana.

Esto produce distorsiones en el espectro que originan componentes espectrales que no existían en la señal original. La Energía de estos componentes espectrales se derrama hacía los costados, provocando deformación, alteración y discontinuidad en los bordes dando lugar a problemas de resolución y dispersión espectral.

Hasta ahora se ha estado usando implícitamente una ventana rectangular, porque multiplica por uno la señal, siendo ésta el tipo de ventana más simple. Pero con este tipo de ventana es como más se sufren estos efectos. En la Figura 3.9 se puede observar el espectro de frecuencias de una ventana rectangular correspondiente a un intervalo determinado de la señal de entrada. En él se puede ver como hay un lóbulo central perteneciente a la frecuencia fundamental, y lóbulos laterales producidos por el efecto ventana.

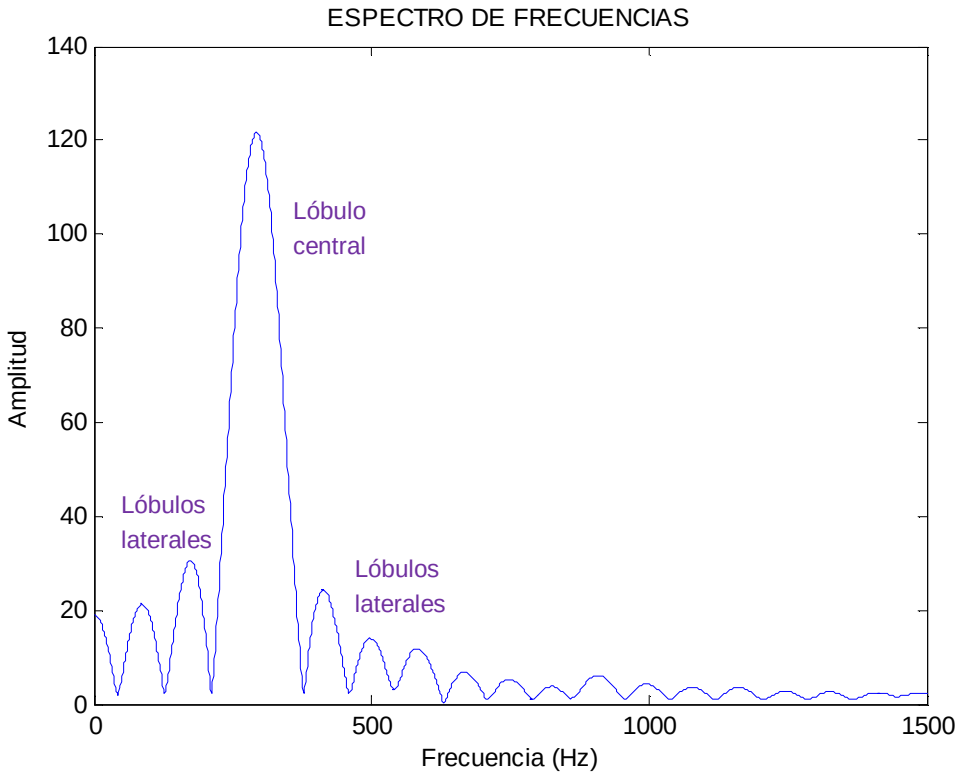


Figura 3.9 Espectro de frecuencias usando una ventana rectangular.

Para minimizar el efecto ventana hay que reducir los lóbulos laterales y para ello se debe cambiar la forma de la ventana. Las ventanas no rectangulares, tratan de corregir este defecto y las que tienen forma de campana dan buenos resultados. La ventana Hamming es una buena elección ya que tiene buena resolución en frecuencia y en amplitud.

Referente al tamaño de la ventana, existe un caso particular cuando el tamaño de ésta es múltiplo del período de la señal original. En tal caso se obtiene un espectro ideal ya que se muestra el espectro continuo en los puntos de discontinuidad y el efecto ventana es nulo. El problema es que el período de la señal puede ir cambiando al ritmo que cambian las notas por lo que la ventana debería ser de diferente tamaño en cada nota y eso es algo que no se puede saber de antemano ya que un músico puede cambiar de partitura, estar componiendo, realizar una improvisación, etc.

Descartada la opción anterior, se podría considerar hacer la ventana más grande para obtener un mejor resultado de la FFT. Así que, y teniendo en cuenta que para el cálculo de la FFT en el procesado de datos es mejor que la longitud sea potencia de dos, y siendo la longitud calculada inicialmente de 535 muestras, ésta pasaría a ser de 1024 muestras.

En la Figura 3.10 se representa el espectro de frecuencias del mismo intervalo pero utilizando diferentes ventanas. La Figura 3.10 A pertenece a una ventana rectangular. En la Figura 3.10 B se utiliza una ventana Hamming de 535 muestras de longitud donde se puede apreciar que los lóbulos laterales han desaparecido pero el lóbulo central se ha ensanchado y su amplitud ha disminuido. Por último, en la Figura 3.10 C se utiliza también una ventana Hamming pero de 1024 muestras, donde el lóbulo central se ha estrechado y su amplitud ha aumentado levemente respecto la original.

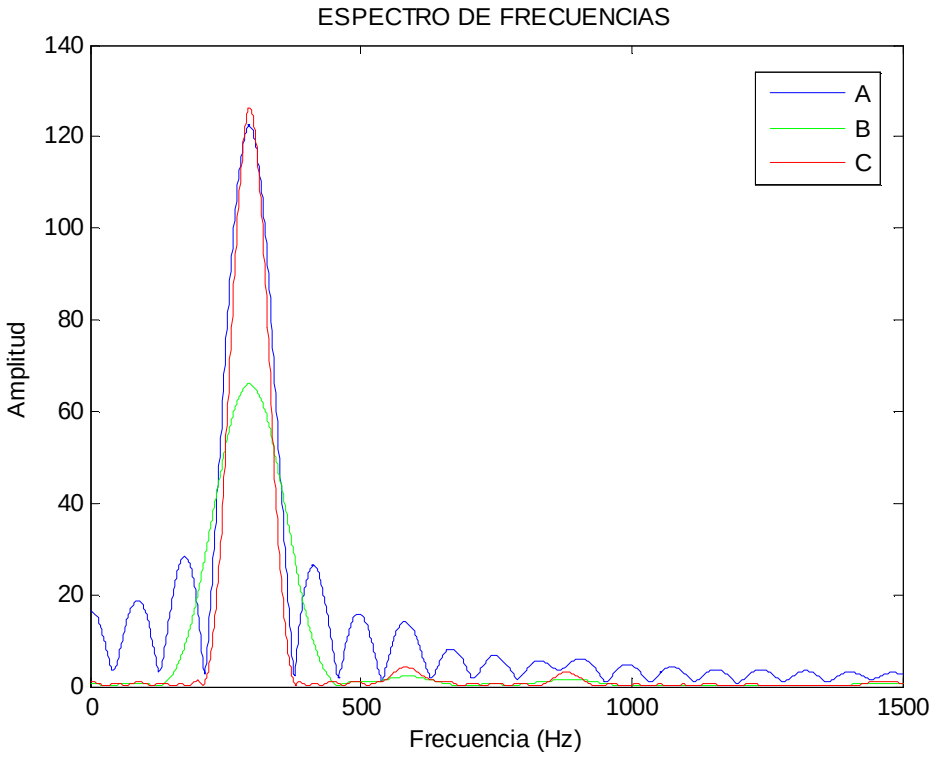


Figura 3.10 Espectro de frecuencias usando varias ventanas. (A) Ventana rectangular de 535 muestras. (B) Ventana Hamming de 535 muestras. (C) Ventana Hamming de 1024 muestras.

Para cambiar el tipo de ventana hay que multiplicar los datos de la ventana muestra a muestra por unos coeficientes, ya que por defecto es como si estuvieran multiplicados por uno. MATLAB posee la función *window()* que permite generar un vector con estos coeficientes según el tipo de ventana y de su longitud.

Siguiendo con el ejemplo de la Figura 3.7 y la Figura 3.8, en la Figura 3.11 puede observarse la notable mejora del uso de una ventana rectangular de 535 muestras (A) respecto una ventana con forma de campana de 1024 muestras (B) donde la variación de la frecuencia pasa de ser de unos 18 Hz a una variación de 2 Hz manteniéndose incluso estable en algunos intervalos de tiempo.

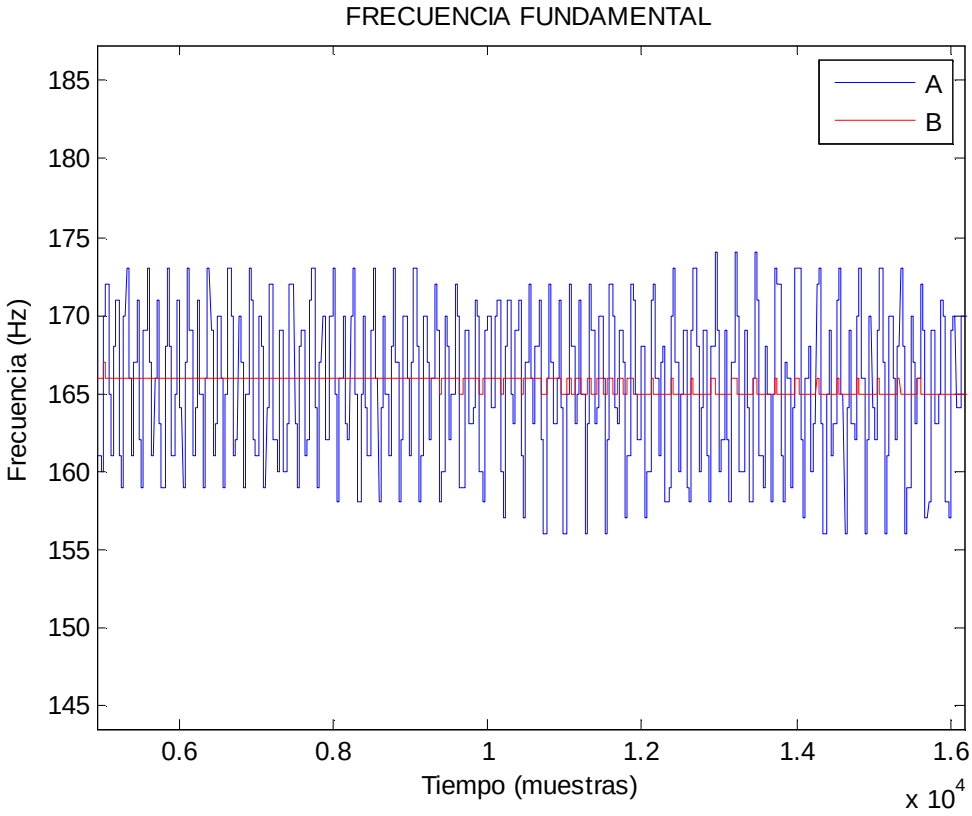


Figura 3.11 Zoom de la frecuencia fundamental correspondiente a la nota Mi_3 de 164,8 Hz usando varias ventanas. (A) Ventana rectangular de 535 muestras. (B) Ventana Hamming de 1024 muestras.

Por lo tanto, para reducir el efecto ventana y mejorar el espectro de frecuencias, la ventana debe tener forma de campana, por ejemplo del tipo Hamming, y una longitud de 1024 muestras.

Filtraje de la frecuencia fundamental

Otra forma de solucionar el problema del efecto ventana es filtrando la frecuencia sin necesidad de cambiar el tipo ni la longitud de ventana.

Una opción es usar un filtro de media móvil con pesos exponenciales también llamado filtro discreto pasa-bajos de primer orden muy utilizado en procesos de control. Es un filtro computacionalmente eficiente que únicamente necesita el valor de la frecuencia filtrada anteriormente y el valor de la frecuencia actual sin filtrar. Su expresión matemática viene dada por la ecuación 3.8.

$$\hat{y}[k] = \lambda \cdot \hat{y}[k-1] + (1-\lambda) \cdot y[k] \tag{3.8}$$

Donde $\hat{y}$ es la frecuencia fundamental filtrada, y es la frecuencia fundamental sin filtrar y λ es un valor constante entre 0 y 1 que indica el nivel de filtrado. Si $\lambda = 1$ el filtrado es excesivo y el filtro no tiene el efecto deseado, y si $\lambda = 0$ el filtro no tiene ningún efecto. Para calcular λ se aplica la ecuación 3.9 y para calcular (1-λ) la ecuación 3.10.

$$\lambda = \frac{N_L}{N_L+1} \tag{3.9}$$

$$(1-\lambda) = \frac{1}{N_L} + 1 \tag{3.10}$$

N_L es la longitud de la ventana, por lo que si la ventana es de 535 muestras, $\lambda = 0{,}9981$.

Otra opción es utilizar un filtro FIR (Finite Impulse Response) que es un filtro muy utilizado en el procesado digital de señales. Es siempre estable y se basa en la entrada actual y en las *n* entradas anteriores donde éstas coinciden con el orden del filtro y con el número de coeficientes. Su expresión matemática viene dada por la ecuación 3.11.

$$\hat{y}[n] = \sum_{k=0}^{N-1} h[k] \cdot y[n-k] \tag{3.11}$$

Donde $\hat{y}$ es la frecuencia fundamental filtrada, y la frecuencia fundamental sin filtrar, N el orden del filtro, h los coeficientes del filtro y n la entrada actual. MATLAB permite generar un vector con los coeficientes del filtro mediante la función *fir1()*.

Siguiendo con el ejemplo de la Figura 3.7 y la Figura 3.8, en la Figura 3.12 se ha hecho un zoom donde se puede ver como también mejora la señal usando filtros. Con el filtro de media móvil (B) la variación de frecuencia ha pasado de ser de unos 18 Hz a 2 Hz con una $\lambda = 0{,}9$, ya que con $\lambda = 0{,}9981$ se filtra demasiado y el resultado no es bueno ya que está muy próxima a 1, por lo que se ha reducido donde un valor de λ entre 0,85 y 0,95 también da buenos resultados. Con el filtro FIR (C) la variación de

frecuencia ha pasado de ser de unos 18 Hz a 1 Hz para un orden de N = 25 sin mostrar apenas oscilación teniendo en cuenta que, cuanto mayor sea el orden, mejor es el filtrado pero también mayor tiempo computacional, empezando a ser bueno a partir de orden 16.

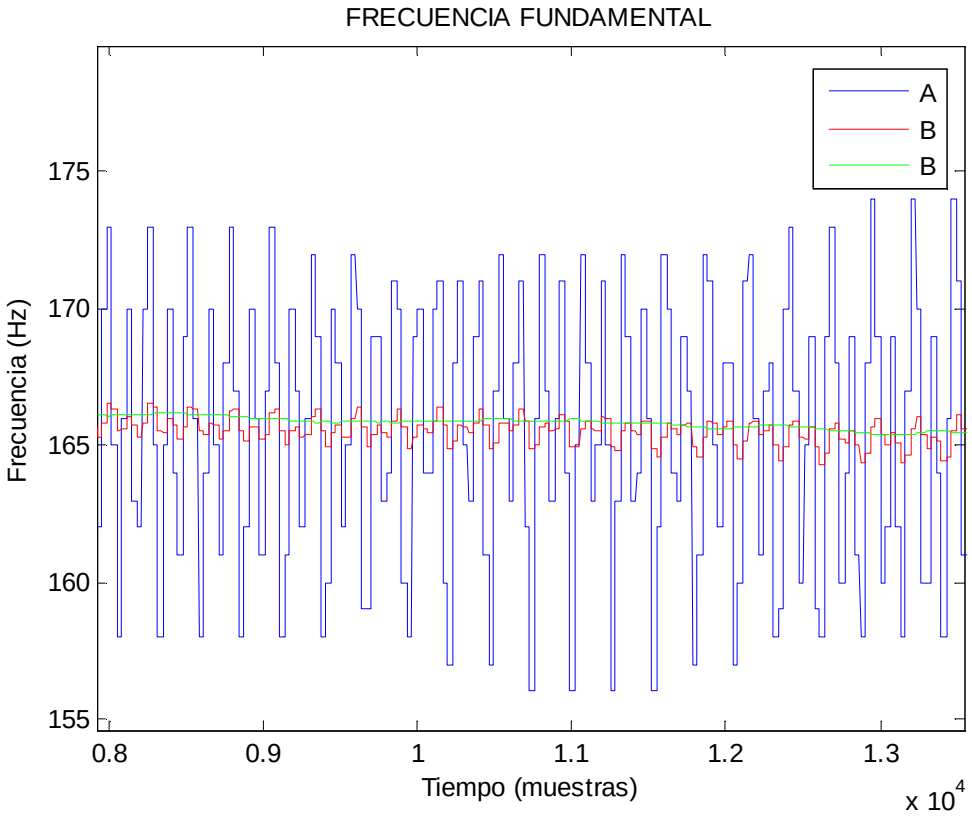


Figura 3.12 Zoom de la frecuencia fundamental correspondiente a la nota Mi_3 de 164,8 Hz usando filtros. (A) Sin filtro. (B) Filtro de media móvil con pesos exponenciales. (C) Filtro FIR.

Establecer la frecuencia de la nota

Tanto cambiando la ventana como usando filtros, la frecuencia sigue teniendo una pequeña oscilación. Se puede ir un poco más lejos haciendo las dos cosas, es decir, primero se cambia la ventana y luego se aplica el filtro, de esta manera se reduce al máximo esta oscilación siendo esta de 0,5 Hz aproximadamente. Aunque también se puede dar un paso más para no sufrir definitivamente el efecto ventana y asegurar que la frecuencia no oscile, y es establecer un valor fijo e invariable de frecuencia mientras dure la nota.

Para ello, la nota debe permanecer como mínimo durante un tiempo determinado dentro de un margen de frecuencias, por lo que se deberán realizar un número mínimo de FFT's. Si esto se cumple, la frecuencia queda establecida con un único valor y no cambiará mientras permanezca dentro de ese margen. Pero si por cualquier motivo no se cumple (ruido, cambio de nota, picos de frecuencia, etc.) la frecuencia no se puede establecer. En este último caso el valor de la frecuencia queda indefinido mientras ésta

no se establece, por lo que conviene asignar mientras tanto, el valor de la frecuencia establecida anteriormente.

La Figura 3.13 muestra el diagrama de flujo relacionado con el algoritmo que establece la frecuencia.

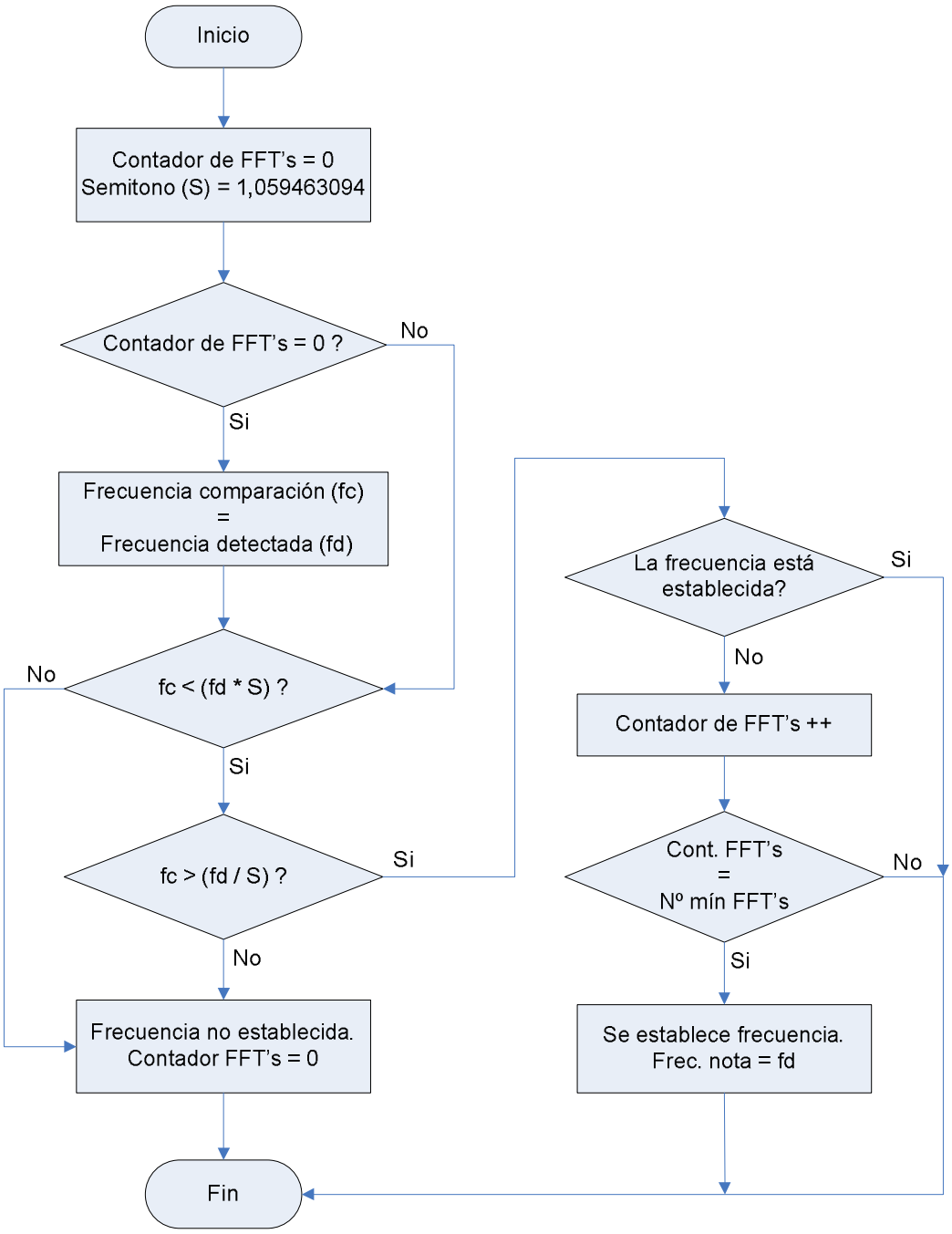


Figura 3.13 Diagrama de flujo del algoritmo para establecer la frecuencia de una nota.

Para establecer la frecuencia se deben realizar un número determinado de FFT's, por lo que cada vez que se realice una FFT se incrementará un contador. Inicialmente este contador vale cero y la frecuencia detectada se establece como frecuencia inicial

o de comparación. A partir de ese momento, cada vez que se detecte una frecuencia se comparará con la inicial.

El margen de frecuencias está entre la frecuencia de un semitono antes y un semitono después de cada nota detectada. Si la frecuencia inicial está dentro de este margen, el contador se incrementa y así sucesivamente con cada nueva frecuencia que se detecte hasta llegar a un número determinado de FFT's. Cuando se llega a este número se establece la frecuencia de la nota con el último valor detectado. Si la frecuencia ya ha sido establecida, no se incrementa el contador ni se mira si se ha llegado al número mínimo de FFT's ya que no es necesario.

En caso de que la nueva frecuencia detectada no esté dentro del margen de frecuencias, el contador se inicializa a cero y se vuelve a empezar todo el proceso de nuevo con el valor actual de frecuencia. Normalmente esto querrá decir que se está produciendo un cambio de nota.

```
contFFT = 0; % Contador de veces que se realiza la FFT.
semitono = 2^(1/12); % Distancia mínima entre dos semitonos.

frec = frecFundFil2; % Frecuencia fundamental después de filtrar.

%Se establece la frecuencia inicial de comparación:
if contFFT==0
    frecFundInicial = frec;
end
% Se mira si la frecuencia actual respecto a la inicial esta dentro
% del margen valido de frecuencias:
if frecFundInicial<frec*semitono && frecFundInicial>frec/semitono
    if frecEst==0
        % Si esta dentro, se incrementa el contador y se mira
        % si han pasado el numero de FFT para establecer la
        % frecuencia de la nota:
        contFFT = contFFT+1;
        if contFFT==numFFT
            frecNota = frec;
            frecEst = 1;
        end
    end
% Si no está dentro se vuelve a poner el contador de FFT a cero y
% la nota queda como no establecida:
else
    contFFT = 0;
    frecEst = 0;
end
```

Figura 3.14 Algoritmo MATLAB para establecer la frecuencia de una nota.

El hecho de no poder determinar la frecuencia de la nota hasta que pasan un número determinado de FFT's supone un retraso que depende del número de veces que se deba hacer la FFT y del tiempo entre una FFT y otra, es decir, del paso de ventana. En la Figura 3.15 se puede ver el resultado de aplicar el algoritmo de la Figura 3.14 donde no existen picos de frecuencia y cada tramo es fijo y constante. Para ello han sido necesarias 18 FFT's para establecer la frecuencia de la nota por lo

que si teniendo en cuenta que el paso son 0,758 milisegundos, se genera un retraso de 13,6 milisegundos.

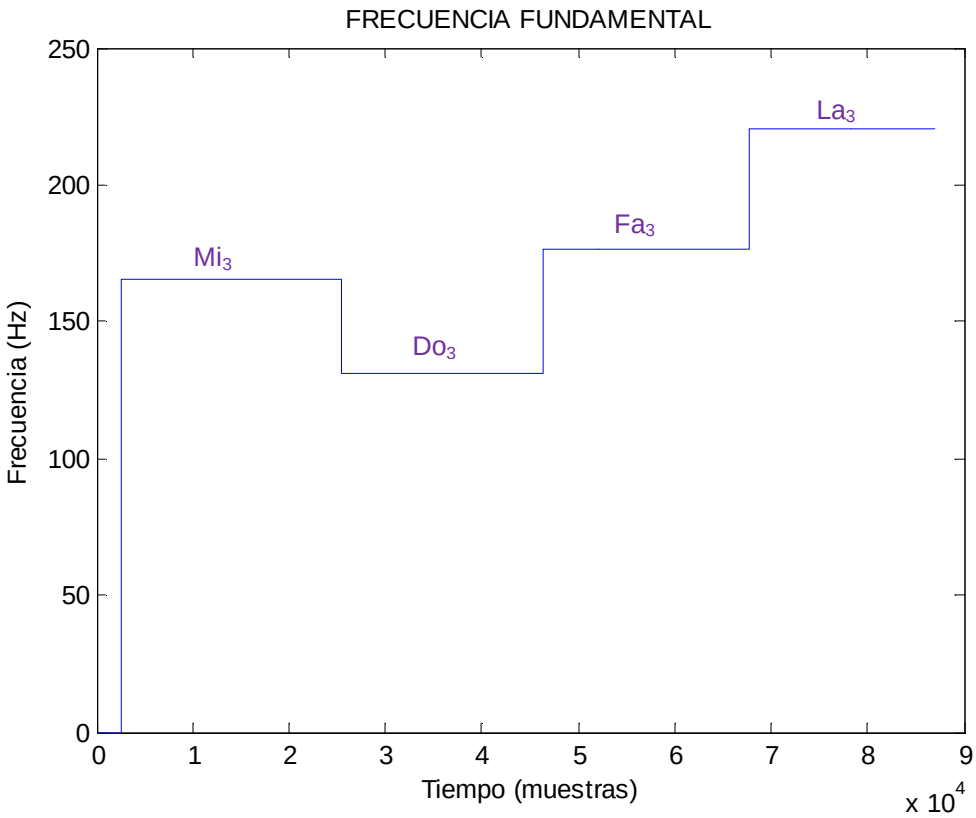


Figura 3.15 Frecuencia fundamental para la secuencia de notas Mi_3-Do_3-Fa_3-La_4 aplicando el algoritmo para establecer notas.

3.1.4 Generar la señal de salida

Una vez se tienen los parámetros correspondientes a la frecuencia fundamental (f_0) y a su amplitud (A_0), y solucionado el problema del efecto ventana, ya se puede determinar la señal de salida y crear una armonía.

Bypass

Una forma de comprobar que todo el proceso y que los cálculos se han hecho correctamente es reproduciendo la misma señal de entrada usando estos dos parámetros. El hecho de reproducir la misma señal de entrada se conoce con el nombre de *bypass* donde no se realiza ningún efecto sonoro ni se genera ninguna armonía. Pero hay que tener en cuenta que como sólo se utiliza la frecuencia fundamental, sin armónicos, la señal de salida tendrá el sonido de un tono puro.

Para cada muestra de la señal de entrada se calcula la señal de salida mediante la ecuación 3.12 y como resultado se genera un vector de salida de *N* muestras.

$$salida_{Bypass}[k] = A_0 \cdot \sin(2\pi \cdot f_0 \cdot t) \tag{3.12}$$

Donde *t* es el tiempo y se puede determinar con la ecuación 3.1 para cada valor de *k* (muestra del instante actual) en vez de *N* (total de muestras).

En la Figura 3.16 se pueden ver la señal de entrada (arriba) y la señal de salida (abajo) donde esta última no es exactamente igual debido a que son tonos puros, pero a pesar de ello, trata de mantener la forma por lo que con la amplitud de la señal a la frecuencia fundamental (A_0) se consigue la envolvente de cada nota.

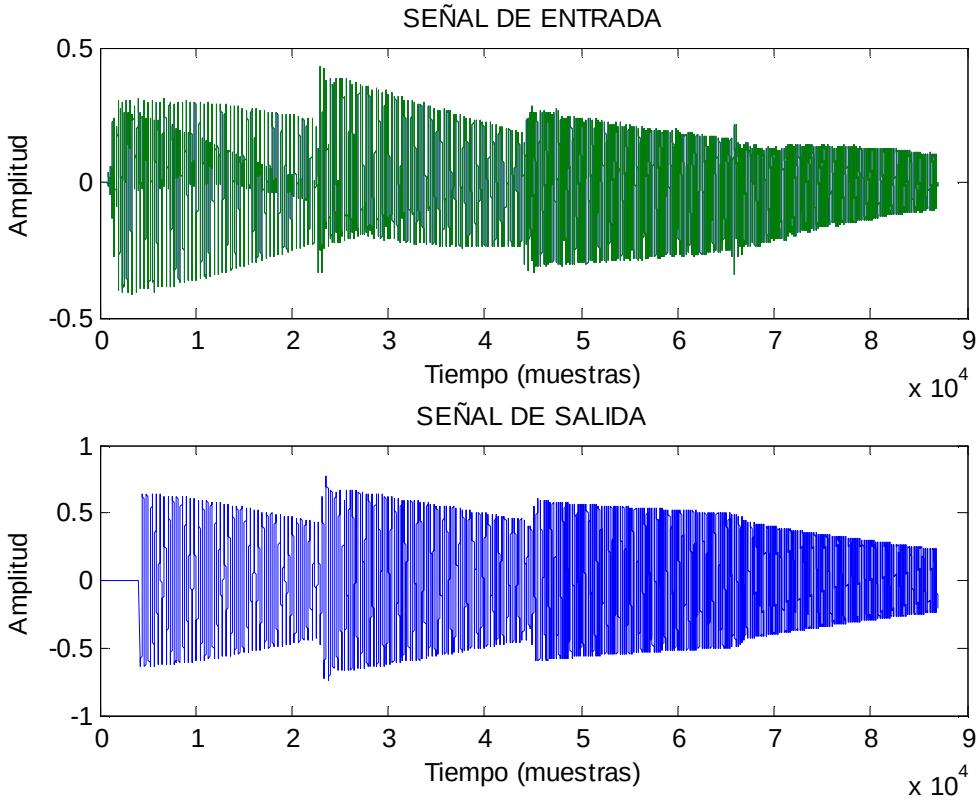


Figura 3.16 Señal de entrada y de salida de la secuencia de notas Sol_2-Do_3-Fa_3-Si_3.

El resultado se guarda en un archivo WAV para poder comprobar que se escucha correctamente. En el caso de generarse el vector de salida sin establecer la frecuencia de la nota se podría notar que el sonido no es bueno, siendo éste peor si no se usa una ventana en forma de campana y/o no se filtra la frecuencia, donde se podría comprobar de forma audible la influencia del efecto ventana.

Acorde

Para formar un acorde se necesitan al menos tres notas donde cada una tiene una frecuencia que se determina en base a la frecuencia de la nota que se detecta. Pero no basta con sólo saber esta frecuencia sino también se necesita saber que nota es, es decir, si es un *Do*, un *Re*, un *Mi*, etc. ya que en función de esa nota se determina la distancia en semitonos del resto de notas para formar el acorde ya que éste depende de la escala y de la tonalidad.

Para saber que nota es, se lleva la frecuencia detectada a la más baja posible por octavas según el instrumento de pruebas, que en este caso es la guitarra. Por ejemplo, si la frecuencia detectada es de 391,9 Hz (Sol_4) ésta quedaría en 97,9 Hz (Sol_2). Entonces se recorre una tabla partiendo de la nota más baja (Mi_2 para la guitarra), que sería la primera posición, hasta llegar a la frecuencia que se ha detectado y modificado (Sol_2), que sería la cuarta posición (Mi_2-Fa_2-$Fa\#_2$-Sol_2). De este modo, se pueden tener tabuladas las notas con sus intervalos en función de esta posición.

Para hacer las pruebas se han cogido los intervalos para formar los acordes de la escala mayor natural en la tonalidad de *Do*. En la Tabla 3.1 se muestra la relación que hay entre la posición de la nota para la octava más baja y las notas para formar acordes con sus intervalos correspondientes. Si se da el caso de tocar una nota que no está en la escala, el acorde no existe, por lo que los intervalos son cero ya que no hay notas que puedan formarlo.

Posición	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Nota	**Mi_2**	**Fa_2**	**$Fa\#_2$**	**Sol_2**	**$Sol\#_2$**	**La_2**	**$La\#_2$**	**Si_2**	**Do_3**	**$Do\#_3$**	**Re_3**	**$Re\#_3$**
1er intervalo	3	4	0	4	0	3	0	3	4	0	3	0
Terceras	Sol	La	-	Si	-	Do	-	Re	Mi	-	Fa	-
2º intervalo	7	7	0	7	0	7	0	6	7	0	7	0
Quintas	Si	Do	-	Re	-	Mi	-	Fa	Sol	-	La	-
3er intervalo	10	11	0	10	0	10	0	10	11	0	10	0
Séptimas	Re	Mi	-	Fa	-	Sol	-	La	Si	-	Do	-

Tabla 3.1 Intervalos para formar acordes de la escala mayor natural en la tonalidad de *Do*.

Así por ejemplo, si se toca la nota Sol_4 a 391,9 Hz primero se cambia a la nota Sol_2 para poderla detectar en la tabla y después se busca el acorde que se forma, que al encontrarse en la cuarta posición es el *Sol-Si-Re* en el caso de una tríada con intervalos de 4 y 7 semitonos, o *Sol-Si-Re-Fa* en el caso de una cuatríada con intervalos de 4, 7 y 10 semitonos.

Entonces, una vez ya se tienen los intervalos del acorde y utilizando la frecuencia fundamental de la nota detectada como frecuencia de referencia, ya se pueden calcular las frecuencias del resto de notas de dicho acorde.

En la Figura 3.17 está representado el diagrama de flujo del algoritmo que determina las frecuencias de los acordes, donde el proceso es el que se describe a continuación.

Primero se establece como frecuencia mínima de comparación (f_{comp}) la frecuencia que hay entre la frecuencia de la nota más baja del instrumento y la frecuencia de la nota anterior. Así, por ejemplo, para la guitarra estaría entre 82,406889 Hz y

77,781746 Hz siendo esta de 80,0943175 Hz. Este valor se utiliza tanto para determinar la frecuencia de la nota en la octava más baja (f_{Low}) como para determinar la posición de la tabla para saber los intervalos de cada nota del acorde.

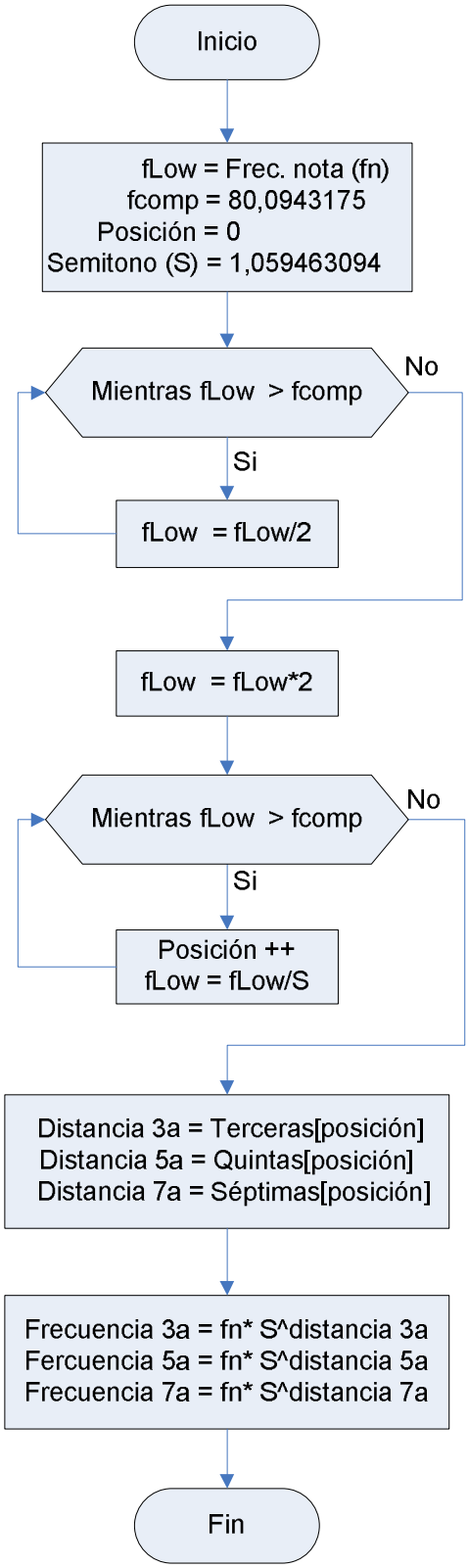


Figura 3.17 Diagrama de flujo del algoritmo para determinar las frecuencias de un acorde.

Después, para calcular f_{Low} se utiliza la ecuación 2.3 de forma reiterada mientras la frecuencia sea mayor que f_{comp}. Como se hace una división de más en la última comparación, se usa la ecuación 2.2 para dejar la frecuencia dentro de la octava más baja. Después se vuelve a comparar nuevamente f_{Low} con f_{comp} de manera que mientras f_{Low} sea mayor que f_{comp} se incrementa una variable al mismo tiempo que se va reduciendo f_{Low} a la frecuencia correspondiente a un semitono menos. Cuando f_{Low} sea menor que f_{comp} el valor de la variable indicará la posición en la tabla para saber la distancia en semitonos de cada intervalo para formar el acorde.

Finamente para determinar la frecuencia de cada nota del acorde se emplea la ecuación 2.1, donde la frecuencia de referencia es la frecuencia fundamental de la nota detectada.

El algoritmo de este proceso está representado en la Figura 3.18.

```
fLow = frecNota; % Frecuencia de la nota establecida.
% Se cambia el valor de la frecuencia de la nota al de su
% octava más baja posible para la guitarra:
while fLow>fComp
    fLow = fLow/2;
end
fLow = fLow*2;
posicion = 0; % Índice de los vectores para las distancias de
              % tereceras,quintas y séptimas.
% Se busca la ubicación de la nota dentro del grupo de notas de
% frecuencia más baja para la guitarra:
while fLow> fComp
    posicion = posicion+1;
    fLow = fLow/semitono;
end
% Distancia en semitonos respecto la nota establecida:
distTercera = tercerasEscala(posicion); % para la tercera
distQuinta = quintasEscala(posicion);   % para la quinta
distSeptima = septimasEscala(posicion); % para la séptima
% Se calcula la frecuencia de cada nota del acorde:
f1 = frecNota;                          % Frecuencia de la nota
f3 = frecNota*semitono^distTercera;   % Frec. de su terceta.
f5 = frecNota*semitono^distQuinta;    % Frec. de su quinta.
f7 = frecNota*semitono^distSeptima;   % Frec. de su séptima.
```

Figura 3.18 Algoritmo MATLAB para determinar las frecuencias de un acorde

Con las frecuencias de cada acorde ya se puede calcular la señal de salida. Esta señal está formada por la suma de varias señales donde cada una corresponde a una nota del acorde. La ecuación 3.13 pertenece a un acorde tríada y la ecuación 3.14 a un acorde cuatríada.

$$salida_{Tríada}[k] = nota + tercera + quinta \tag{3.13}$$

$$salida_{Cuatríada}[k] = nota + tercera + quinta + séptima \tag{3.14}$$

Donde la señal $nota$ corresponde a la misma nota que se toca representada por la ecuación 3.12, por lo que sería equivalente a la $salida_{Bypass}$, y el resto de señales son:

$$tercera = A_0 \cdot \sin(2\pi \cdot f_{3a} \cdot t) \tag{3.15}$$

$$quinta = A_0 \cdot \sin(2\pi \cdot f_{5a} \cdot t) \tag{3.16}$$

$$séptima = A_0 \cdot \sin(2\pi \cdot f_{7a} \cdot t) \tag{3.17}$$

Donde f_{3a} , f_{5a} y f_{7a} son las frecuencias de la tercera, quinta y séptima respectivamente y A_0 es la amplitud a la frecuencia fundamental de la nota detectada.

En la Figura 3.19 se pueden ver las frecuencias de varios acorde cuatríada formados a partir de la secuencia de notas Mi_3-Do_3-Fa_3-La_3.

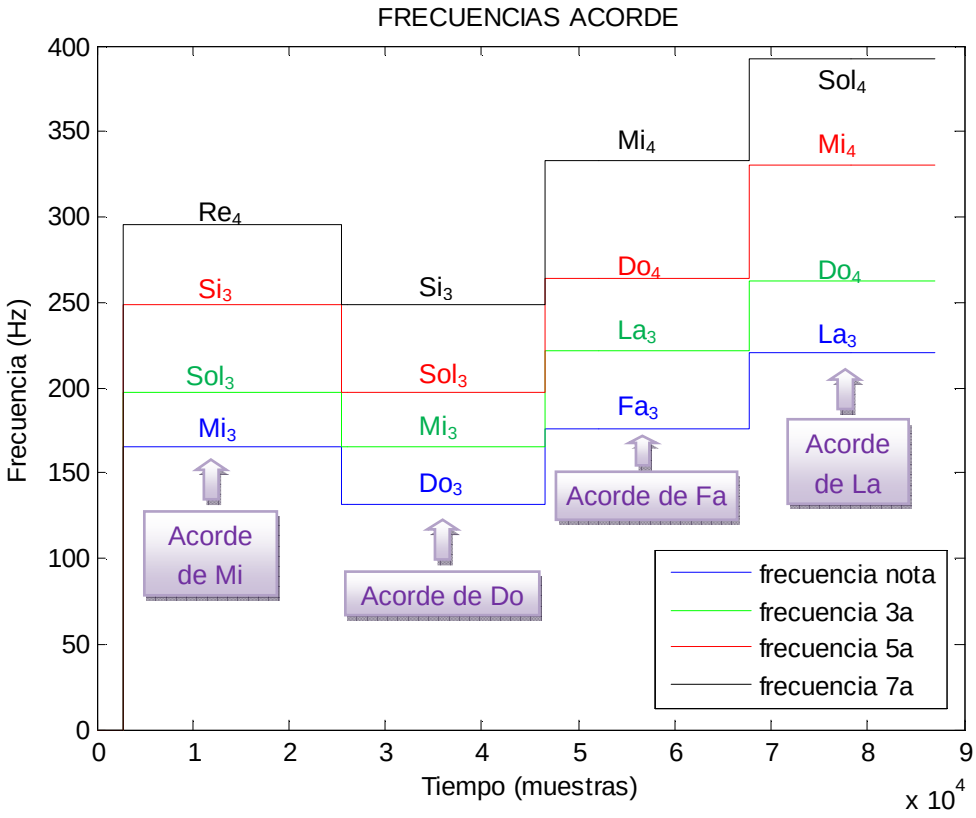


Figura 3.19 Frecuencias de acordes cuatríada para la secuencia de notas Mi_3-Do_3-Fa_3-La_3 en la escala mayor natural de *Do*.

3.1.5 Resultados y conclusiones de la simulación

Mediante MATLAB se ha podido estudiar y desarrollar los pasos y algoritmos necesarios para poder generar una armonía y más concretamente acordes tríada y cuatríada. Para poder generar otro efecto, como un intervalo armónico el planteamiento es el mismo y sólo cambia el número de señales que se deben sumar para generar la señal de salida donde cada una tendrá una frecuencia diferente, como por ejemplo, en un octavador habrá dos notas donde la frecuencia de la segunda será el doble de la primera, en unas quintas la distancia entre la primera y la segunda nota será de siete semitonos, etc.

Resultados

En la Tabla 3.2 están los resultados obtenidos después de realizar pruebas con diferentes archivos WAV con una y varias notas. Para cada nota del archivo se indica el valor de la frecuencia teórica y el de la experimental, tanto de la propia nota o bypass, como del resto de notas que forman el acorde cuatríada. Los resultados del acorde tríada serían igual que el del acorde cuatríada pero omitiendo la columna de la séptima (7ª). El valor teórico corresponde al calculado con la ecuación 2.1 cogiendo como frecuencia de referencia el La_4 a 440 Hz y el valor experimental es el obtenido mediante MATLAB. Por último, también se indica en la tabla la diferencia de frecuencias entre el valor teórico y el experimental.

Como se puede observar, a pesar de que se evita que la frecuencia oscile y se fija para que sea estable, los resultados no coinciden con el valor teórico aunque se aproxima bastante. Esto entre otras cosas es debido a que las señales no han sido grabadas directamente con una guitarra, ya que para las pruebas con MATALB han sido generados archivos WAV con el software Guitar Pro y éste crea las notas de forma sintética o artificial.

Esta diferencia entre la frecuencia teórica y la detectada experimentalmente varía entre 0,2 Hz y 1,9 Hz para la frecuencia de la primera nota o de bypass correspondiente a la frecuencia fundamental detectada. Después se puede observar que como esta nota ya tiene un error, el cálculo del resto de frecuencias de las notas del acorde también lo tendrán ya que su frecuencia se calcula a partir de la primera. Este error aumenta a la vez que aumenta la distancia entre las notas, siendo la máxima variación de 3,2 Hz.

De todas formas, esta variación de frecuencia respecto la teórica es aceptable si se tiene en cuenta que entre dos notas consecutivas hay una margen de frecuencia, y que éste es mayor cuanto más grande sea la frecuencia de la nota. Para el caso más desfavorable de la guitarra la nota más baja es la Mi_2, donde la variación entre esta nota y la siguiente es de 4,9 Hz. Para esta nota, en la simulación se ha obtenido una variación de 1,5 Hz por lo que se estaría dentro del margen.

Notas archivos WAV	Frec. teórica (Hz)				Frec. experimental (Hz)				Δf (Hz)			
	Bypass	3ª	5ª	7ª	Bypass	3ª	5ª	7ª	Bypass	3ª	5ª	7ª
Do_3	130,8	164,8	195,9	246,9	131,2	165,4	196,6	247,7	0,4	0,6	0,7	0,8
$Do\#_3$	138,5	-	-	-	139,2	-	-	-	0,7	-	-	-
Do_4	261,6	329,6	391,9	493,8	262,8	331,1	393,7	496,1	1,2	1,5	1,8	2,3
Re_3	146,8	174,6	220	261,6	148,4	176,5	222,4	264,4	1,6	1,9	2,4	2,8
					296	352	443,5	527,4	##	##	##	##
$Re\#_3$	155,5	-	-	-	156,2	-	-	-	0,7	-	-	-
Re_4	293,6	349,2	440	523,2	295,5	351,4	442,7	526,4	1,9	2,2	2,7	3,2
Mi_2	82,4	97,9	123,4	146,8	168,9	200,9	253,1	300,9	##	##	##	##
					83,9	99,8	125,8	149,6	1,5	1,9	2,4	2,8
Mi_3	164,8	195,9	246,9	293,6	166,1	197,5	248,9	296	1,3	1,6	2	2,4
Mi_4	329,6	391,9	493,8	587,3	330,5	393	495,2	588,9	0,9	1,1	1,4	1,7
Mi_5	659,2	783,9	987,7	1174,6	659	783,7	987,4	1174	0,2	0,2	0,3	0,6
Fa_3	174,6	220	261,6	329,6	175,9	221,6	263,5	332,1	1,3	1,6	1,9	2,5
$Fa\#_3$	184,9	-	-	-	186,2	-	-	-	1,3	-	-	-
Fa_4	349,2	440	523,2	659,2	350,5	441,6	525,2	661,7	1,3	1,6	2	2,5
Sol_2	97,9	123,4	146,8	174,6	198,7	250,4	297,8	354,1	##	##	##	##
Sol_3	195,9	246,9	293,6	349,2	197,1	248,3	295,3	351,1	1,2	1,4	1,7	1,9
$Sol\#_3$	207,6	-	-	-	208,3	-	-	-	0,7	-	-	-
Sol_4	391,9	493,8	587,3	698,4	393,1	495,3	589	700,4	1,2	1,5	1,7	2
La_2	110	130,8	164,8	195,9	219,7	261,3	329,2	391,5	##	##	##	##
$La\#_2$	116,5	-	-	-	117,2	-	-	-	0,7	-	-	-
La_3	220	261,6	329,6	391,9	219,3	260,8	328,6	390,7	0,7	0,8	1	1,2
$La\#_3$	233	-	-	-	234	-	-	-	1	-	-	-
Si_2	123,4	146,8	174,6	220	124,2	147,7	175,6	221,3	0,8	0,9	1	1,3
Si_3	246,9	293,6	349,2	440	247,9	294,9	350,7	441,8	1	1,3	1,5	1,8
Silencio	0	-	-	-	0	-	-	-	0	-	-	-

Tabla 3.2 Resultados de la simulación con MATLAB.

Esto quiere decir que el resto de notas también estarán dentro del margen ya que éste se va haciendo más grande a medida que la nota sube de tono. Por ejemplo, la máxima variación que se ha dado es de 3,2 Hz que corresponde con la frecuencia de la nota Do_5 a 523,2 Hz donde el margen con la nota siguiente es 31,1 Hz.

Otro detalle que se puede observar es que hay unos resultados, indicados en color rojo, que son de aproximadamente el doble de lo que debería ser, y esto es debido a que la frecuencia detectada es la perteneciente al segundo armónico, ya que tiene el doble de frecuencia del primero, y ocurre porque tiene una amplitud mayor que el fundamental.

Hay otros casos, indicado en color naranja, donde el resultado cambia a lo largo del tiempo. Al principio corresponde aproximadamente con el teórico pero al cabo de un tiempo los valores pasan a ser el doble, debido a lo mismo que el caso anterior. También ocurre lo contrario, es decir, al principio el valor de frecuencia es el doble y luego pasa a ser parecido al teórico. Esto es debido a que la amplitud de los armónicos no es siempre igual en todas las fases de la envolvente o del ADSR de la nota, originando en algún momento de esta fase que el segundo armónico tenga más amplitud que el primero.

En el caso de haber un silencio queda comprobado que la frecuencia se mantiene a cero y no se genera ningún acorde. Tampoco se crea ningún acorde en las notas que están fuera de la escala mayor natural de *Do*, que es caso de las notas con sostenidos, donde sólo se detecta la frecuencia fundamental correspondiente a la nota del archivo, pero no se genera ninguna nota más.

Conclusiones

Con MATLAB se ha podido modificar el tamaño de la ventana en el momento de hacer la FFT para tener una resolución de 1 Hz. En la implementación con DSP puede que no sea posible por lo que habrá que tener en cuenta la frecuencia de muestreo con la que se puede trabajar y la longitud de ventana que se vaya a utilizar.

La frecuencia que se obtiene con el algoritmo de detección varía con un margen de 18 Hz debido al efecto ventana. Para solucionarlo existen diferentes alternativas. Por un lado se puede modificar la ventana cambiándola por una con forma de campana con una longitud de 1024 muestras reduciendo este margen a 2 Hz. Por otro lado, se puede filtrar la frecuencia llegando a reducirlo hasta 1 Hz en el caso de un filtro FIR de al menos orden 25. También se puede cambiar la ventana y además filtrar la frecuencia donde se puede conseguir reducirlo la oscilación hasta 0,5 Hz. Por último, aplicando un algoritmo para estabilizar la frecuencia se puede eliminar completamente la oscilación, pero para ello se deben realizar al menos 18 FFT originando un pequeño retraso. En cualquier caso siempre será mejor usar también una ventana con forma de campana para que el resultado de la FFT sea mejor y obtener así un valor más

parecido al teórico. Con MATLAB se pueden generar los coeficientes necesarios tanto para cambiar la forma de la ventana como para usar el filtro FIR.

En el algoritmo de detección no es necesario recorrer todo el vector del resultado de la FFT. Con que se recorran las posiciones referentes al rango de frecuencias que se pretende analizar ya es suficiente y se ahorra tiempo computacional. El problema de este algoritmo, y esto será una cuestión importante a la hora de implementarlo con el DSP con un instrumento real, es que sólo es efectivo cuando la frecuencia fundamental tiene una amplitud mayor que el resto de armónicos y se mantiene así a lo largo del tiempo.

La variación de frecuencia obtenida respecto el valor teórico está entre 0,2 y 3,2 siendo ésta aceptable dado que a frecuencias bajas el margen entre dos notas consecutivas es como mínimo de 4,9 Hz.

Durante los silencios aparecen frecuencias no deseadas debidas al ruido. Este problema se ha solucionado viendo el valor de la amplitud de la señal de entrada antes de realizar la FFT.

Para determinar las frecuencias de las notas de un acorde se han tabulado las distancias de los intervalos entre la nota detectada y el resto de notas del acorde, siendo estas igual a cero si no existe la nota en la escala. Para ello se aplica un algoritmo que determina la posición en la tabla a partir de la frecuencia detectada. La tabla que se ha creado sirve sólo para la escala mayor de *Do* por lo que para utilizar otras escalas y otras tonalidades serán necesarias otras tablas. Finalmente la señal de salida de un acorde se consigue con la suma de las señales individuales de cada nota que forman el acorde.

3.2 Implementación con un DSP

En este apartado se va a desarrollar la aplicación con un DSP teniendo en cuenta el trabajo realizado en el apartado de simulación por lo que ya hay mucho terreno ganado. El objetivo es el mismo que en simulación pero trabajando en tiempo real y utilizando como señal de entrada, la de un instrumento en lugar de una señal sintética grabada en un archivo de audio.

Al principio se realizarán pruebas con señales correspondientes a tonos puros creados con el generador de funciones ya que como sólo tienen un armónico facilitará la detección y análisis de la frecuencia fundamental. Posteriormente, se utilizará una guitarra eléctrica, para hacer pruebas con un instrumento real.

El planteamiento es el siguiente:

a) Determinar la frecuencia de muestreo y el tamaño de la ventana para trabajar en tiempo real.
b) Adquisición de la señal de entrada.
c) Filtrado de la señal de entrada y decimado de la frecuencia de muestreo.
d) Cálculo de la FFT.
e) Determinar la frecuencia fundamental y amplitud máxima.
f) Determinar efectos y acordes.
g) Generar la señal de salida.

El primer paso que se va a realizar es determinar la frecuencia de muestreo y el tamaño de ventana para trabajar en tiempo real y tener una resolución frecuencial y temporal aceptables.

Después, se trata la forma en que se realiza la adquisición de datos mediante el códec de audio y los periféricos McBSP y EDMA. La CPU, en última instancia filtra la señal y decima la frecuencia de muestreo para cumplir con las especificaciones de resolución determinadas anteriormente.

El siguiente paso es la detección de la frecuencia fundamental y su amplitud. Para ello se utiliza el mismo proceso que en simulación donde se aplica el mismo algoritmo. Los coeficientes para aplicar una ventana en forma de campana se obtienen con MATLAB y la función que realiza el cálculo de la FFT se obtiene de las librerías del software de desarrollo.

Posteriormente se analizan los resultados y se mejora el algoritmo de detección para poderlo usar con un instrumento real, ya que el de simulación sólo resulta efectivo con tonos puros.

Finalmente se tratan los efectos y se genera la señal de salida. Según el efecto se determinan el número de notas y sus frecuencias, y en el caso de los acordes se tiene en cuenta la escala y la tonalidad.

En la Figura 3.20 se puede ver el diagrama de flujo de la aplicación DSP donde después de los procesos de configuración e inicialización, el programa se queda en un bucle infinito a la espera de que la CPU sea interrumpida. En este programa sólo se produce un tipo de interrupción y es cuando se ha completado la adquisición de un paquete de datos.

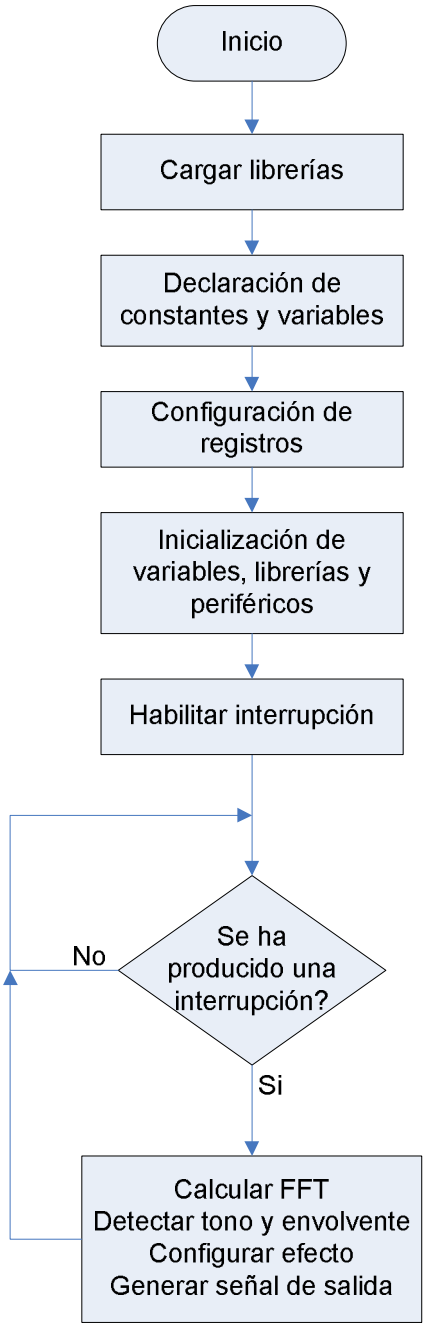


Figura 3.20 Diagrama de flujo de la aplicación DSP.

3.2.1 Frecuencia de muestreo y ventana

El primer, paso antes de empezar a procesar los datos es determinar la frecuencia de muestreo (*Fs)* y la longitud de ventana (N_L) que se van a utilizar con el DSP. En la simulación, como se trabajaba con una *Fs* de 44100 Hz, se cambiaba el tamaño de la ventana de 1024 a 44100 muestras con MATLAB para tener una frecuencia de 1 Hz, pero el hecho de agrandar la ventana en el cálculo de la FFT implicaba un tiempo computacional muy elevado.

Trabajando en tiempo real esto supone un problema, ya que una ventana tan grande, aunque mejora la resolución frecuencial, empeora la resolución temporal originando dos posibles problemas. El primero, es que se podrían capturar varios eventos temporales (más de una nota) que supondrían una superposición de frecuencias en el espectro, y el segundo, es que supondría un mayor tiempo computacional por lo se podría perder la siguiente información a procesar. Con MATLAB esto no ocurría ya que los datos a analizar ya se encontraban guardados en un vector, de forma que, el programa esperaba a terminar el cálculo de la FFT de una ventana antes de hacer el cálculo de la siguiente ventana, el único problema era que había que esperar.

A parte de tener en cuenta estos problemas temporales, también hay que considerar otros aspectos relacionados con el tamaño de la ventana trabajando con un DSP, como son el número de ciclos que necesita la CPU para calcular la FFT, y la relación señal/ruido de la señal de entrada. Además, también es importante establecer una *Fs* de trabajo, la cual determinará la frecuencia máxima que se puede detectar.

Ciclos CPU

En función del tamaño de la ventana, la CPU tarda más o menos tiempo en calcular la FFT. Aplicando la ecuación 3.18 se pueden determinar los ciclos que necesita la CPU de la placa de desarrollo para calcular la FFT.

$$ciclos = log_2(N_L) \cdot \left(4 \cdot \frac{N_L}{2} + 7\right) + 34 + \frac{N_L}{4} \tag{3.18}$$

Así, por ejemplo, para una N_L de 1024 muestras tarda 20840 ciclos y para una N_L de 2048 muestras tarda 45679 ciclos. Teniendo en cuenta que la CPU trabaja con una frecuencia de 225 MHz suponen 0,09 ms y 0,2 ms respectivamente. Por lo que aumentar el doble el tamaño de la ventana supone más del doble del tiempo de cálculo de la CPU.

Relación señal/ruido

Antes de calcular la FFT, es importante dividir todos los datos entre la longitud de la ventana para evitar que se desborde el resultado. Esto supone reducir el valor de la

señal de entrada afectando al rango dinámico inicial haciendo peor la relación señal/ruido. El rango dinámico se puede calcular aplicando la ecuación 3.19.

$$Rango\ dinamico = 20 \cdot \log\left(2^{nbits}\right) \tag{3.19}$$

Como los datos son de 16 bits, el rango dinámico es de 96 dB, para calcular la FFT correctamente trabajando con este procesador, hay que aplicar la ecuación 3.20.

$$Rango\ dinamico = 20 \cdot \log\left(\frac{2^{nbits}}{N_L}\right) \tag{3.20}$$

Por lo tanto para una ventana de 1024 muestras el rango dinámico es de 36 dB y para una de 2048 es de 30 dB. Esto quiere decir que cuanto mayor es la ventana el resultado se puede ver más afectado por el ruido u otras señales como los propios armónicos de la frecuencia fundamental, haciendo que sea más difícil detectarla.

Frecuencia máxima

Con una *Fs* de 44100 Hz y aplicando la ecuación 3.7 se puede detectar una frecuencia máxima de 22050 Hz. Debido a que sólo se pretende detectar la frecuencia fundamental, este valor de *Fs* es excesivo teniendo en cuenta que en la mayoría de instrumentos la frecuencia fundamental máxima está por debajo de 2000 Hz, como en el caso de la guitarra donde la frecuencia fundamental máxima es de 1319 Hz.

Resolución frecuencial y temporal

Teniendo en cuenta las consideraciones anteriores, y que el margen de frecuencias más pequeño entre dos notas para una guitarra es de 4,9 Hz, la solución pasa por cambiar la *Fs*, ya que una *Fs* de 44100 Hz supone una resolución frecuencial de 43 Hz para una ventana de 1024 muestra, o de 21,5 Hz para una de 2048.

Aplicando las ecuaciones 3.1, 3.4 y 3.7 se puede determinar la resolución temporal, la resolución frecuencial y la frecuencia máxima respectivamente para diferentes valores de *Fs* y tamaños de ventana:

- Si se impone una frecuencia máxima de 2000 Hz aplicando la ecuación 3.7 resultaría una *Fs* de 4000 Hz.
- Para una resolución frecuencial máxima de 4,9 Hz y una *Fs* de 4000 Hz, aplicando la ecuación 3.4, la ventana debería ser como mínimo de 817 muestras de longitud, pero como tiene que ser potencia de dos para calcular la FFT, ésta se cambia a 1024 con lo que se mejora la resolución bajando a 3,9 Hz (3,90625 Hz).
- Finalmente, aplicando la ecuación 3.1, la resolución temporal es de 256 ms, siendo éste el intervalo de tiempo de la señal que coge la ventana, por lo que durante este tiempo, no puede haber más de una nota.

Esto quiere decir que se podría detectar desde la nota Do_2 de 65,4 Hz hasta la nota Si_6 de 1975,5 Hz y se podrían tocar cuatro notas por segundo. De esta manera quedan predefinidas algunas limitaciones de la aplicación.

En único inconveniente es que la *Fs* más baja del DSP es de 8000 Hz y cambiar a este valor de *Fs* implicaría doblar el tamaño de la ventana para mantener la resolución frecuencial y temporal. Pero como aumentar el tamaño de la ventana supone principalmente más tiempo computacional y peor rango dinámico o relación señal/ruido, se propone disminuir la *Fs,* o mejor dicho, decimar la *Fs*, para mantenerla a 4000 Hz usando una ventana de 1024 muestras. Esto implicará tener que filtrar la señal de entrada para tener una frecuencia máxima de 2000 Hz.

3.2.2 Adquisición de datos. Códec, McBSP y EDMA

En principio, la adquisición de datos se hace sin la intervención de la CPU mediante los periféricos EDMA, McBSP y el códec de audio que incorpora la placa de desarrollo. En la Figura 3.21 se muestra el diagrama de bloques de cómo se realiza la adquisición de datos.

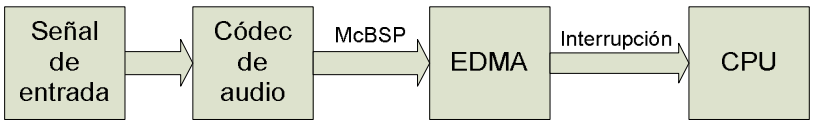


Figura 3.21 Adquisición de datos con el DSP.

Códec de audio

En la primera etapa de la adquisición de datos el instrumento se conecta a la línea de entrada (Line In) del códec donde la señal se codifica para procesarla digitalmente. El códec permite diferentes frecuencias de muestreo tanto para la conversión analógica-digital de la señal de entrada (ADC) como en la conversión digital-analógica de la señal de salida (DAC). En la Tabla 3.3 se muestran las combinaciones disponibles.

	Frecuencia de muestreo (kHz)										
ADC	96	88,2	48	44,1	32	8,021	8	48	44,1	8	8,021
DAC	96	88,2	48	44,1	32	8,021	8	8	8,021	48	44,1

Tabla 3.3 Frecuencias de muestreo del códec de audio.

El códec es estéreo por lo que posee dos canales (izquierdo y derecho). Esto hace que en cada adquisición, las tramas de datos sean de 32 bits, 16 bits para el canal izquierdo y 16 para el derecho. El formato de estos datos son números enteros con signo por lo que las variables del programa para tratar las señales de entrada y salida

son del tipo *short*. Esto quiere decir que los datos de entrada tendrán valores comprendidos entre -32768 y +32767.

Para configurar el códec existen varios registros que permiten gestionar el volumen, el formato de datos, la frecuencia de muestreo, etc. En la Figura 3.22 y en la Figura 3.23 se muestran el esquema de bloques y la configuración del códec respectivamente.

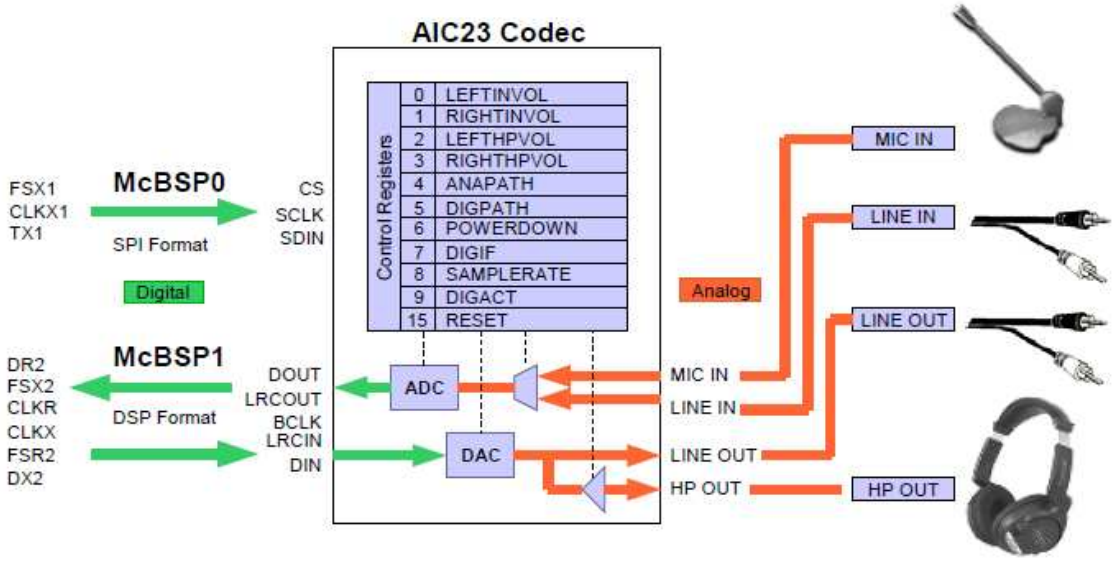


Figura 3.22 Esquema de bloques del códec de audio.

```
DSK6713_AIC23_Config codecConfig = {
0x0017, //0 DSK6713_AIC23_LEFTINVOL Left line input channel volume
0x0017, //1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume
0x01F9, //2 DSK6713_AIC23_LEFTHPVOL Left channel headphone volume
0x01F9, //3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume
0x0011, //4 DSK6713_AIC23_ANAPATH Analog audio path control
0x0000, //5 DSK6713_AIC23_DIGPATH Digital audio path control
0x0000, //6 DSK6713_AIC23_POWERDOWN Power down control
0x0043, //7 DSK6713_AIC23_DIGIF Digital audio interface format
0x000D, //8 DSK6713_AIC23_SAMPLERATE Sample rate control IN/OUT:8KHz
0x0001  //9 DSK6713_AIC23_DIGACT Digital interface activation
};
```

Figura 3.23 Configuración del códec de audio.

McBSP

El McBSP (Multichannel Buffered Serial Ports) es un puerto serie multicanal que dispone de buffers de entrada y salida donde almacena los datos automáticamente, por lo que la transmisión puede ser simultánea. El procesador dispone de dos puertos serie: el McBSP0 que es unidireccional y el McBSP1 que es bidireccional.

La comunicación con el códec de audio se realiza mediante el puerto serie McBSP1 ya que como es un canal bidireccional se pueden enviar datos al códec o recibir datos procedentes del códec.

Por un lado, el códec de audio está conectado al puerto McBSP (Figura 3.22), por lo que cada vez que el códec lee una muestra pasa directamente al puerto McBSP. Por otro lado, la comunicación de este puerto está conectada con el EDMA (Figura 3.24), esto quiere decir que el McBSP enlaza con el EDMA antes de hacerlo con la CPU.

EDMA

El EDMA (Enhanced Direct Memory Access) es un periférico que se encarga de transferir datos entre la memoria o dispositivos externos sin intervención de la CPU. El EDMA trabaja en paralelo con la ejecución del programa de forma que mientras se está realizando la adquisición de datos, la CPU se puede dedicar a otras operaciones haciendo que haya mayor velocidad de procesamiento en el DSP.

En esta aplicación el EDMA se usa dos formas. La primera para la comunicación entre el códec de audio y la CPU en la adquisición de datos, y la segunda moviendo datos en memoria tanto en la adquisición como durante el procesado.

La adquisición de datos se realiza mediante la técnica del doble buffer donde los datos procedentes del códec se guardan en un buffer de entrada mientras se procesan los datos almacenados en el otro buffer. De esta manera se permite realizar simultáneamente los procesos de adquisición y procesado de datos, garantizando la ejecución en tiempo real de la aplicación. Esto da lugar a dos ciclos de trabajo que se van alternando continuamente. En la Tabla 3.4 se muestran las operaciones que realizan el EDMA y la CPU en cada ciclo.

Ciclo	EDMA	CPU
0	Adquisición de datos en bufferIn 1	Procesar datos del bufferIn 2
1	Adquisición de datos en bufferIn 2	Procesar datos del bufferIn 1

Tabla 3.4 Operaciones que realizan el EDMA y la CPU en cada ciclo.

La longitud de los buffers de entrada está relacionada con el paso de ventana. En este caso el paso corresponde con el tiempo que se están adquiriendo muestras con el EDMA mientras se procesan con la CPU los datos adquiridos anteriormente. Hay que decir que en la adquisición de datos el códec lee los dos canales (izquierdo y derecho) independiente de que la entrada se mono o estéreo por lo que la longitud de cada buffer será igual al doble del paso de ventana. Como la señal que genera un instrumento es mono sólo habrá datos en uno de los dos canales y en el otro serán cero.

La siguiente operación que hace el EDMA después de llenar el buffer de entrada es preparar los datos para que sean procesados por la CPU. Como lo primero que se va a hacer es filtrar la señal de entrada, el EDMA coge los nuevos datos que se van a

procesar (los correspondientes a uno de los dos canales) y los coloca junto con las últimas *n* muestras de entrada anteriores correspondientes al orden del filtro.

Finalmente el EDMA interrumpe la CPU para que pueda procesar los datos. En la Figura 3.24 se muestra el proceso que realiza el EDMA en la adquisición de datos.

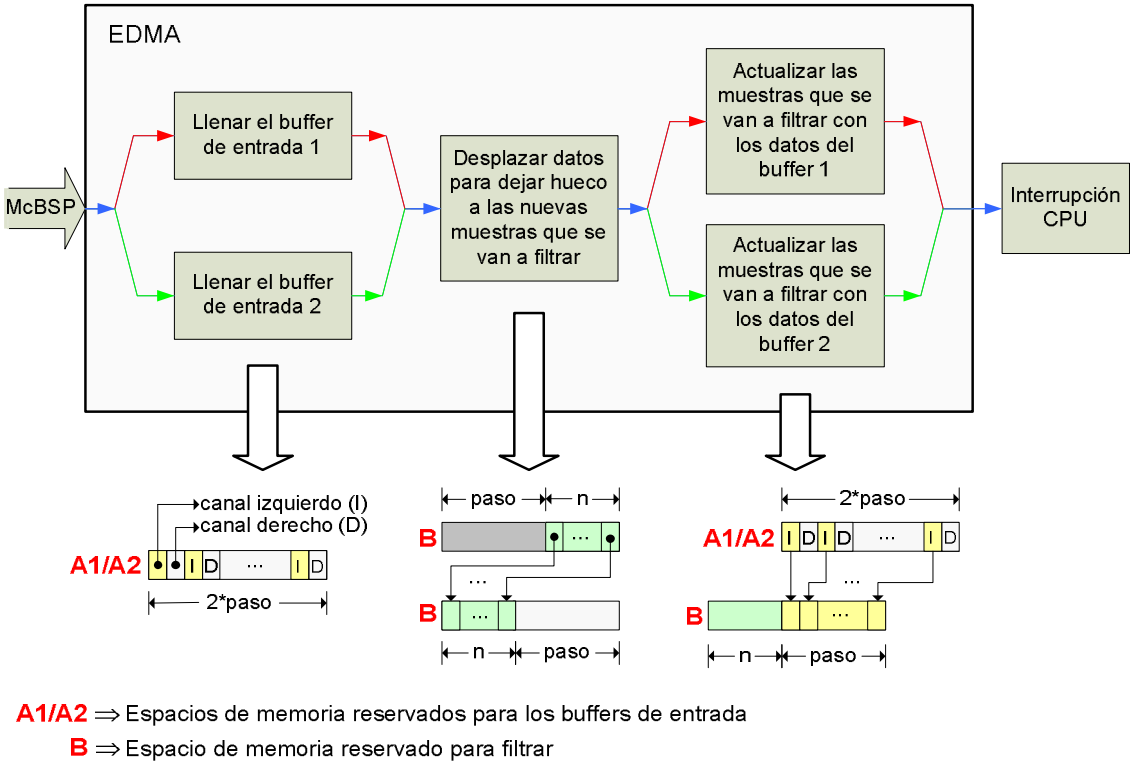


Figura 3.24 Transferencia de datos con el EDMA en la adquisición.

Para mover datos en la memoria o acceder a otros dispositivos el EDMA utiliza diversos canales de acceso que se deben configurar. Para ello tiene unos registros donde se debe indicar para cada canal la dirección de origen, la dirección de destino, la cantidad de datos que se mueven en la transferencia y si los datos se mueven individualmente (elemento) o en paquetes de datos (array/frame). Además, el EDMA dispone de canales de recarga que permiten actualizar un canal de acceso para realizar una nueva transferencia de datos cuando la anterior ya se ha completado como en el caso del doble buffer de entrada.

A parte de configurar los registros, también se ha implementado la función *edam_init()* que se encarga de inicializar el EDMA donde básicamente se realiza:

- Abrir los canales, reiniciarlos y deshabilitarles la interrupción.
- Reservar espacio y asignar las direcciones de recarga.
- Configurar los canales de acceso y los canales de recarga.
- Habilitar los canales.
- Borrar posibles interrupciones pendientes del EDMA.
- Habilitar la interrupción por finalización de lectura del EDMA.
- Habilitar el encadenado de los canales por finalización de lectura del EDMA.

Tanto la configuración de los canales y registros como la implementación de la función de inicialización del EDMA se encuentran con más detalle en el archivo *Armonizador.c* del apéndice D.

Filtraje de la señal de entrada y decimado la *Fs*

Para decimar la *Fs* y pasar de 8000 Hz a 4000 Hz, simplemente hay que coger una muestra de la señal de entrada de cada dos, por lo que de la cantidad de muestras de entrada, correspondientes a un canal y que son igual al paso, sólo se cogen la mitad. Al dividirse entre dos tanto la *Fs* como la cantidad de muestras, se mantiene la misma relación original por lo que se continúa teniendo el mismo margen de tiempo para el procesado.

El inconveniente es que hay que filtrar las muestras para evitar el efecto *aliasing*. Este efecto no deseado es un solapamiento de frecuencias que impiden la posterior reconstrucción correcta de la señal. Para evitarlo hay que aplicar el teorema de Nyquist y para ello la frecuencia de la señal de entrada no debe ser superior a ½ de la *Fs.* La placa de desarrollo tiene filtros antialiasing pero sólo para las *Fs* de que dispone. Por eso se debe diseñar un filtro para que corte las frecuencias a 2000 Hz.

En este caso también se ha usado un filtro FIR como el utilizado en simulación. Para implementarlo se ha aplicado la ecuación 3.11 y para determinar el orden y los coeficientes del filtro se ha utilizado la aplicación *FDAtool* de MATLAB que permite diseñar filtros rápidamente. En la Figura 3.25 aparecen los parámetros de frecuencia y magnitud que se han utilizado para diseñar el filtro.

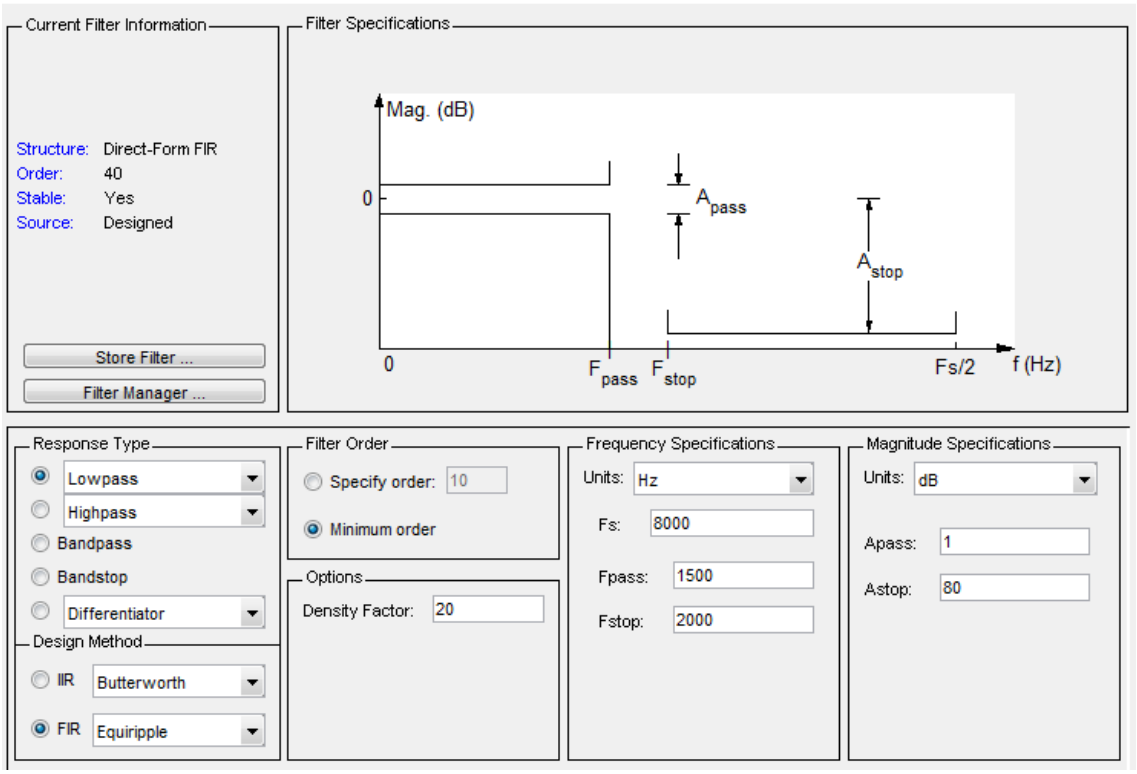


Figura 3.25 Diseño del filtro antialiasing con MATLAB.

Para que no saliera un orden muy elevado debido a una fuerte pendiente de atenuación que supondría más tiempo computacional, se ha puesto una frecuencia de paso de 1500 Hz para una frecuencia corte de 2000 Hz. Esto hace que la nota más alta que se pueda detectar será un $Fa\#_6$ de 1479,9 Hz, ya que, como se puede observar en la Figura 3.26, a partir de 1500 Hz la señal se empieza a atenuar.

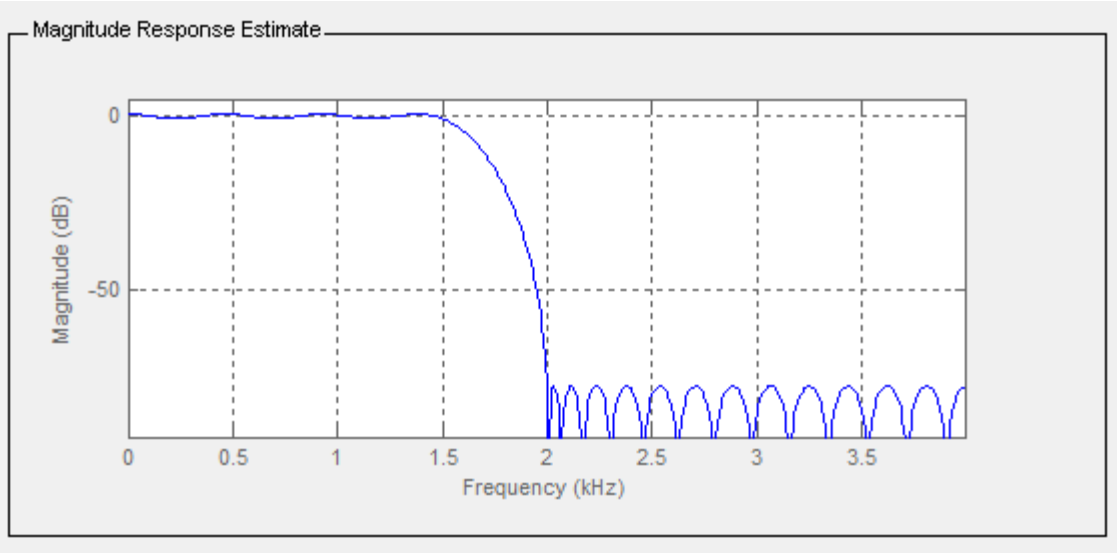


Figura 3.26 Respuesta del filtro antialising.

En la Figura 3.27 se puede ver que la longitud del filtro es de 41 coeficientes, por lo que es de orden n = 41.

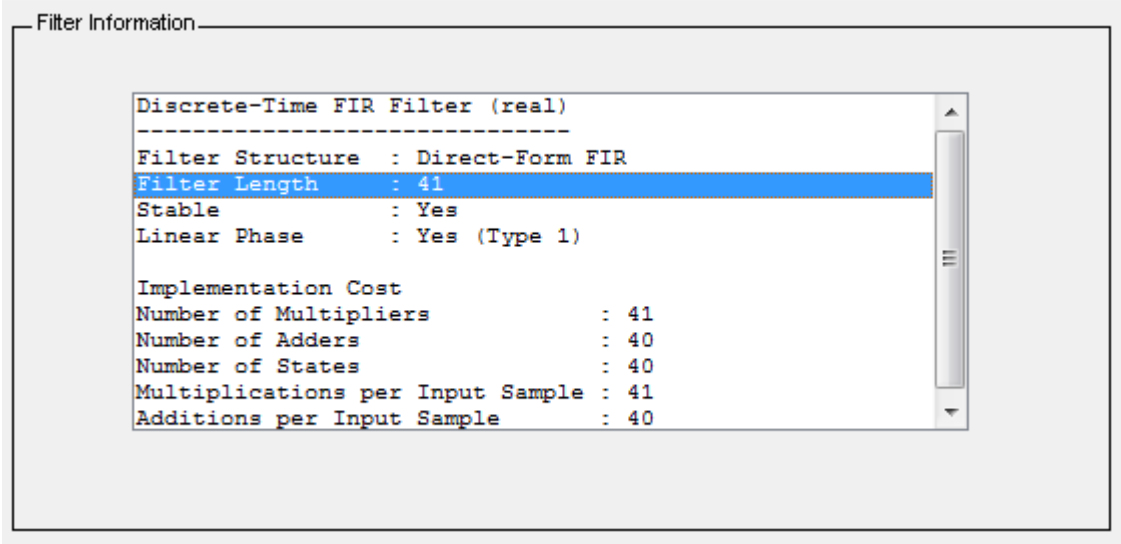
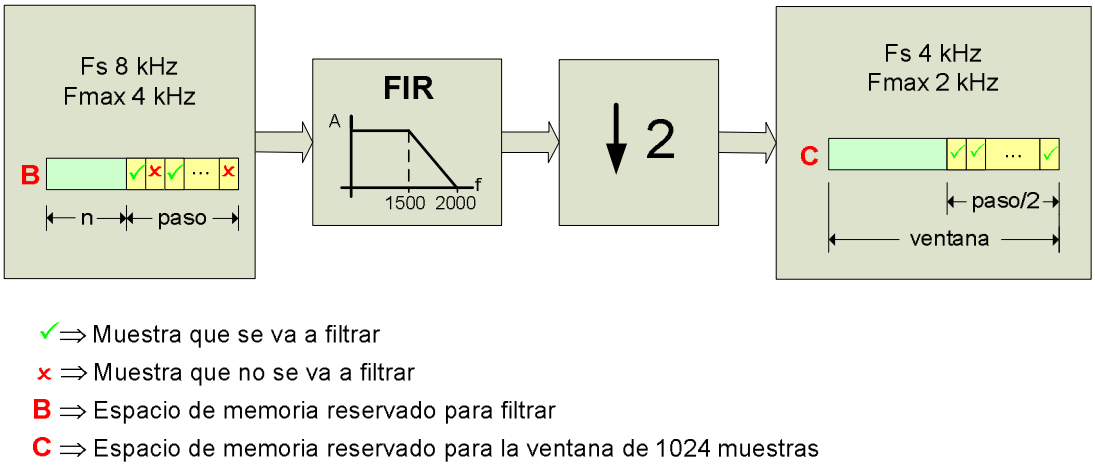


Figura 3.27 Información del filtro antialiasing.

La aplicación *FDAtool* permite exportar los coeficientes a una archivo *.h* Teniendo en cuenta el tipo de datos para poderlos usar con código C. Como las variables que se tratan en la aplicación son del tipo *short*, el tipo de datos son enteros con signo de 16 bits.

En la Figura 3.28 se muestra el proceso de filtrar y decimar donde finalmente los datos pasan a la posición de memoria correspondiente a la ventana que se usará para calcular la FFT. Este proceso lo debe realizar la CPU ya que se realizan operaciones que no puede hacer el EDMA por lo que finalmente la CPU sí que interviene en la adquisición de datos.



✓ ⇒ Muestra que se va a filtrar
x ⇒ Muestra que no se va a filtrar
B ⇒ Espacio de memoria reservado para filtrar
C ⇒ Espacio de memoria reservado para la ventana de 1024 muestras

Figura 3.28 Filtraje de la señal de entrada y decimado de la *Fs*.

3.2.3 Detección de la frecuencia fundamental y amplitud

Los pasos a seguir para detectar la frecuencia fundamental y su correspondiente amplitud son:

a) Cambiar el tipo de ventana a una con forma de campana.
b) Calcular la FFT.
c) Ordenar el resultado.
d) Calcular el módulo.
e) Aplicar el algoritmo de detección para determinar la posición y la amplitud.
f) Calcular la frecuencia fundamental con la ecuación 3.5.

Tipo de ventana

Este paso es importante para obtener un buen resultado de la FFT. Para cambiar la ventana a una con forma de campana, se multiplican las muestras ya filtradas una a una por los coeficientes del tipo de ventana. Gracias a MATLAB se han generado los coeficientes usando la función *window(),* y mediante la función *dlmwrite()* se ha creado un fichero *.h* para poder usarlos con código C. En Figura 3.29 se muestra el código MATLAB.

```
longVentana = 1024;     % Longitud de Ventana
tipoVentana = @hamming; % Tipo de Ventana

% Se llena el vector con los coeficientes de la ventana:
coef = window(tipoVentana ,longVentana);

% Se pasan los datos a short:
for j=1:longVentana
    coefVentana(j) = round(coef(j)*32768);
end

% Se genera un fichero .h para usar los coeficientes con código C:
dlmwrite('hamming.h',coefVentana,',');
```

Figura 3.29 Código MATLAB para generar los coeficientes de la ventana.

Cálculo de la FFT

El software de desarrollo CCS tiene las librerías C62 y C67 que incluyen unas funciones que permiten calcular la FFT. La librería C62 trabaja con datos de coma fija de tipo *short* y la librería C67 con datos de coma flotante de tipo *float* o *double*. Como los datos que trata el códec, tanto en la adquisición como en la salida, son enteros de 16 bits del tipo *short*, resulta más adecuado trabajar con la librería C62. Además usando datos con coma flotante se pierde precisión ya que al pasarlos a enteros se pierde la parte decimal.

La función que realiza el cálculo de la FFT es *DSP_radix2().* Esta función tiene algunos requisitos para que el resultado sea correcto. Algunos de ellos resultarán familiares ya que se han comentando en varias ocasiones. Estos son:

- Que los datos de la ventana sean del tipo *short* con signo de 16 bits.
- Que la longitud de la ventana sea potencia de dos.
- Dividir todos los datos de entrada entre la longitud de la ventana para evitar que se desborde el resultado.

Después de hacer la FFT el resultado queda desordenado por lo que no se puede aplicar la ecuación 3.5 para determinar la frecuencia. Para ordenarlo se usa la función *DSP_bitrev_cplx()* pero antes de aplicar el algoritmo de detección, se debe calcular el módulo ya que el resultado de la FFT está en forma compleja. Como realizar la raíz cuadrada suponen mucho tiempo computacional, se deja el módulo al cuadrado que permite analizar el espectro de la misma forma. En la Figura 3.30 está representado el espectro de una señal correspondiente a un tono puro de 492 Hz donde se puede apreciar que el resultado tiene simetría. Esto quiere decir que para calcular el módulo y determinar la frecuencia fundamental es suficiente con recorrer la mitad del espectro, de esta manera se optimiza el código.

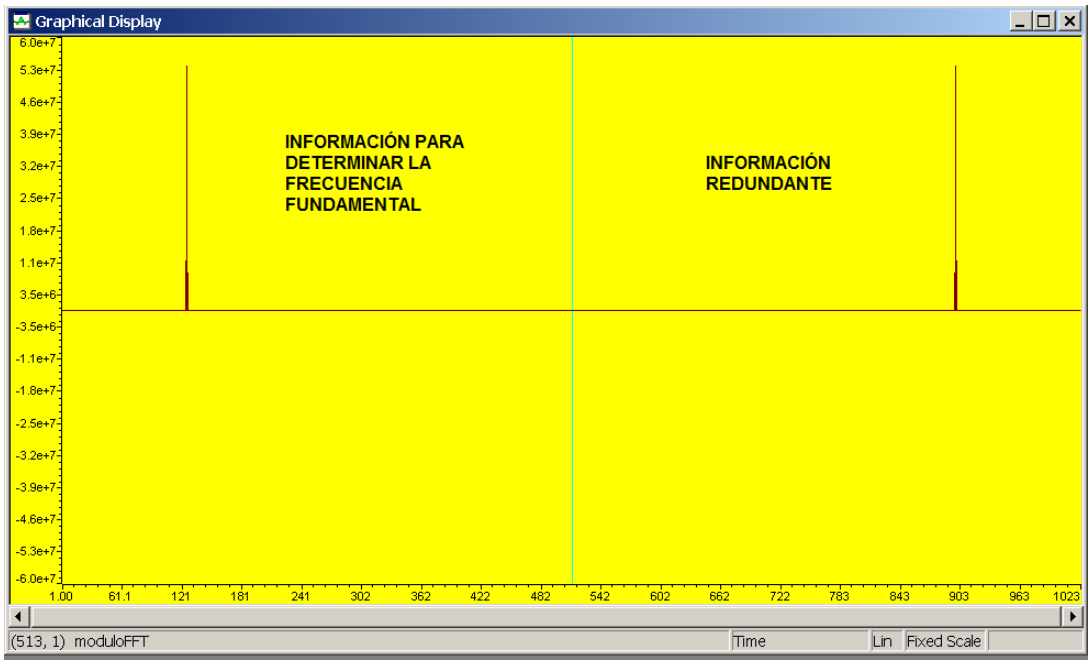


Figura 3.30 Espectro de un tono puro de 492 Hz.

Algoritmo de detección

Se utiliza el mismo algoritmo que en simulación pero teniendo en cuenta dos cosas. La primera es que como la amplitud del espectro corresponde con la del modulo al cuadrado del resultado de la FFT, se debe calcular la raíz cuadrada de la amplitud detectada.

La segunda es que hay que calcular la frecuencia aplicando la ecuación 3.5 con el índice que se obtiene después de aplicar el algoritmo, ya que, a diferencia que en simulación, el índice i correspondía directamente con el valor de frecuencia porque la resolución era de 1 Hz. Por eso el rango de posiciones que se usaba en simulación para la detección estaba comprendido entre la posición 70 y la 1500 para una resolución de 1 Hz, pero ahora, teniendo en cuenta que la resolución es de 3,90625 Hz y que las frecuencias límite de la aplicación son 65 y 1500 Hz, si se aplica la ecuación 3.5 el rango del índice i está comprendido entre las posiciones 16 (i_{min}) y 384 (i_{max}) respectivamente. Pero para no ajustar tanto en los límites se deja una nota más de margen en cada extremo siendo 61,7 Hz el límite por abajo y a 1567,9 Hz el límite por arriba, por lo que i_{min} = 15 y i_{max} = 402. Esto supone que además de optimizar aun más el código, en el caso de aparecer frecuencias no deseadas en el resultado de la FFT, no se tendrían en cuenta porque ya no se miran ni cuando se calcula el modulo del espectro ni cuando se aplica el algoritmo de detección.

En la Figura 3.31 se muestra el algoritmo de detección hecho con el software de desarrollo CCS.

```
amplitudEspectralMax2 = 0; // Se resetea la variable
for (i=iMin; i<iMax; i++) {
    if (moduloFFT[i]>amplitudEspectralMax2) {
        amplitudEspectralMax2 = moduloFFT[i];
    indiceAmplitudEspectralMax = i;
    }
}
// Amplitud espectral máxima:
amplitudEspectralMax = sqrt((double)amplitudEspectralMax2);
// Frecuencia fundamental:
frecuenciaFundamental = (float)indiceAmplitudEspectralMax *
                        resolucionFrecuencia;
```

Figura 3.31 Algoritmo en CCS de detección de la frecuencia fundamental y su amplitud.

A continuación se muestran dos ejemplos gráficos. El primero corresponde al espectro de un tono puro de 1 kHz y el segundo a uno 490 Hz. En cada uno de ellos se ha situado el cursor en la barra con mayor amplitud para indicar la posición en el espectro a la frecuencia fundamental, dato que se puede ver la parte inferior izquierda de la gráfica *(posición, amplitud)*. El primer ejemplo corresponde a la Figura 3.32 y cuando el algoritmo detecta el valor máximo la posición i es igual a 256 por lo que aplicando la ecuación 3.5 para una resolución de 3,90625 Hz corresponde una frecuencia de 1 kHz.

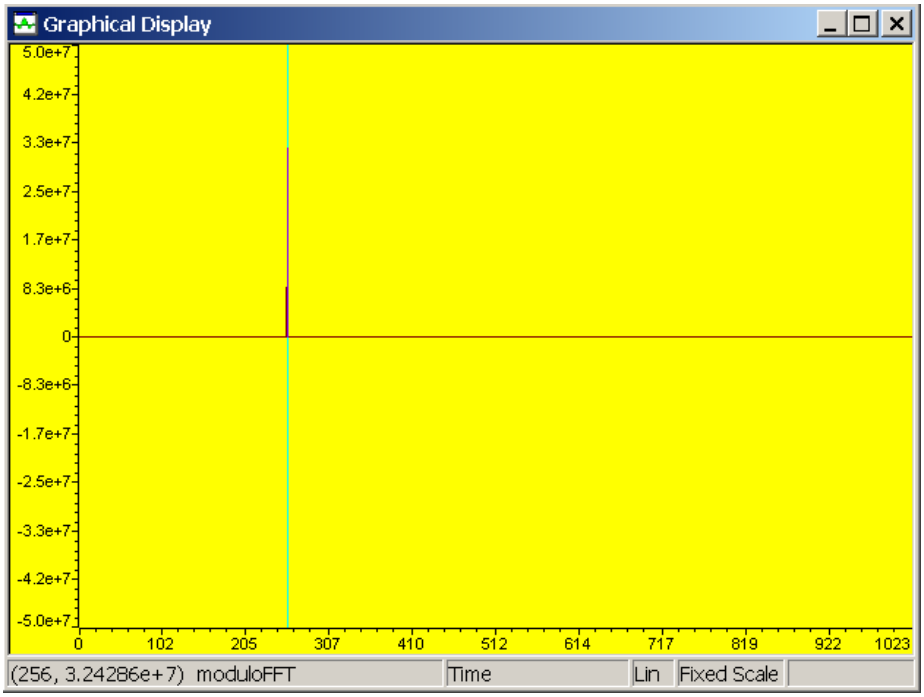


Figura 3.32 Espectro de un tono puro de 1 kHz.

El segundo ejemplo corresponde a la Figura 3.33 y cuando el algoritmo detecta el valor máximo la posición i es igual a 118 por lo que aplicando la ecuación 3.5 para una resolución de 3,90625 Hz corresponde una frecuencia de 490,9375 kHz.

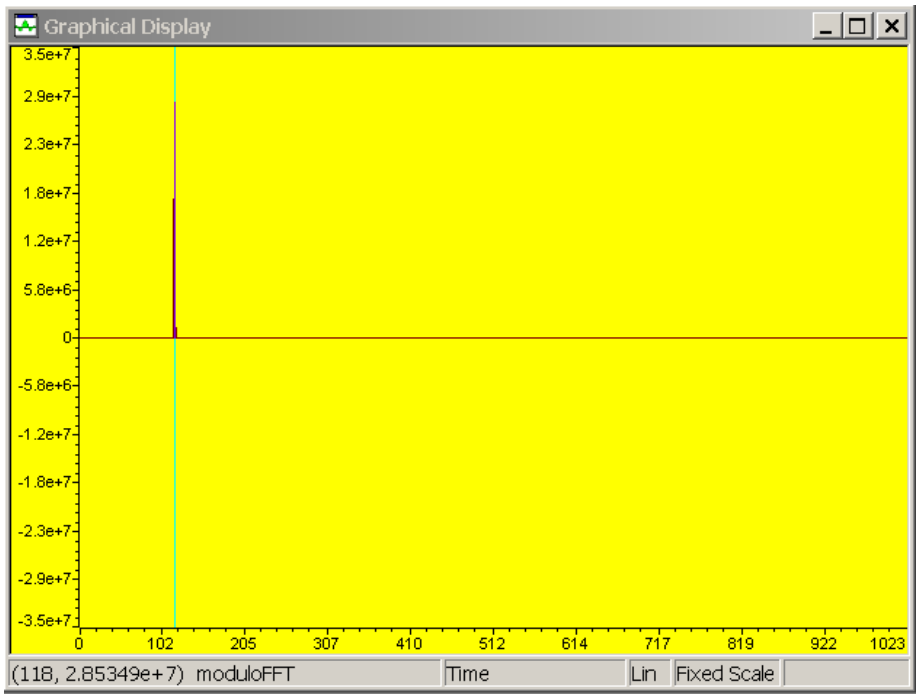


Figura 3.33 Espectro de un tono puro de 460 kHz.

EDMA y siguiente FFT

Después de detectar la frecuencia y su amplitud, ya se pueden preparar los datos de la ventana para el siguiente cálculo de la FFT. Para ello, tal y como se muestra en la Figura 3.34, se deben mover los datos de la ventana para dejar hueco al nuevo paquete de datos de la siguiente adquisición. Para ello se vuelve a usar el EDMA mientras la CPU sigue con el programa.

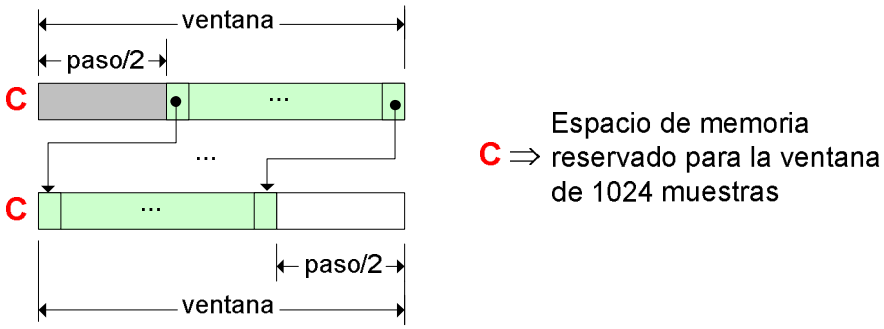


Figura 3.34 Transferencia de datos con el EDMA después de calcular la FFT

3.2.4 Análisis de la frecuencia detectada y su amplitud

Silencios

Como se puede ver en la Figura 3.35 trabajando con el DSP no aparecen frecuencias indeseadas en el espectro cuando hay un silencio o cuando no hay señal. Por lo que, el resultado de la simulación se puede atribuir a la forma de digitalizar los datos que tiene el programa Guitar Pro.

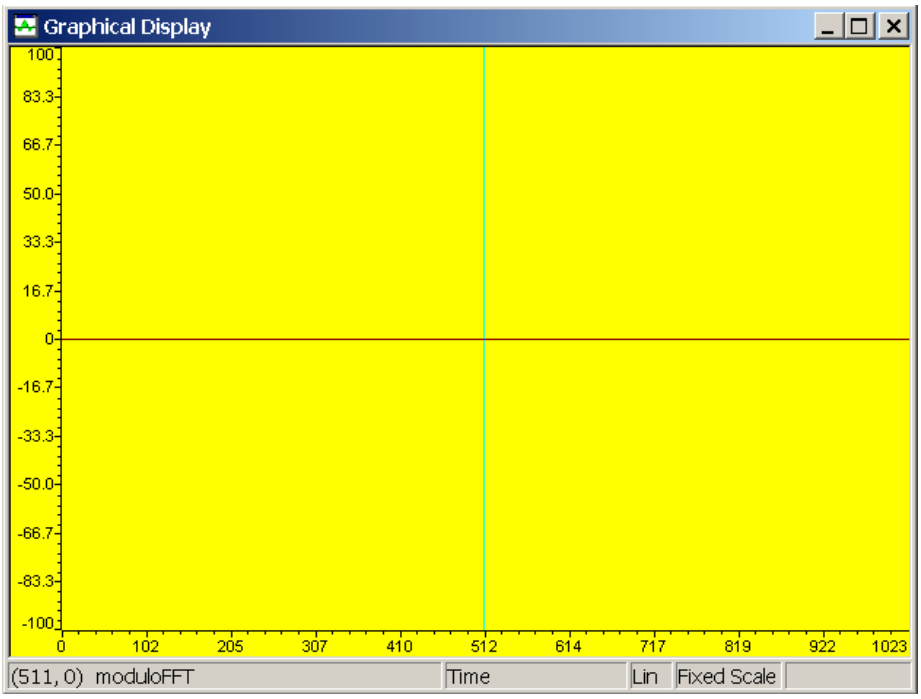


Figura 3.35 Espectro de un silencio.

De esta manera, no es necesario aplicar el algoritmo de detección de silencios y ahorrar el tiempo computacional que supone comparar que todas las muestras de entrada estén dentro de un margen de valores.

Filtraje y establecer frecuencia

Trabajando en simulación, aun cambiando el tipo de ventana y/o filtrando la frecuencia, existía una oscilación que hacía que la frecuencia no fuera estable ya que cada vez que se hacía la FFT la frecuencia fundamental no era exactamente la misma debido al efecto ventana. Esta oscilación era en el mejor de los casos de 0,5 Hz y en el peor de los casos de 2 Hz siendo necesario establecer una frecuencia fija provocando un retraso.

Trabajando con el DSP se ha observado que esto no ocurre y que la frecuencia se mantiene estable en diferentes mediciones a lo largo del tiempo. Esto es debido a que la resolución es de 3,9 Hz y absorbe esta oscilación.

Debido a esto cuando se detecta un armónico no siempre aparece una única barra en el espectro. Dicho de otra forma, si sólo aparece una barra en el espectro, ésta corresponde exactamente a la frecuencia de la señal de entrada, pero si aparecen más barras, la frecuencia real no corresponde a la barra de mayor amplitud sino que se encuentra principalmente entre las dos barras consecutivas más grandes. Esto hace que la frecuencia que se detecte no corresponda exactamente con la de la señal de entrada aunque no está más lejos de 3,9 Hz.

Entonces, la oscilación debida al efecto ventana viene representada por la mayor o menor amplitud de estas dos barras principales donde normalmente siempre una de las dos será la que siempre tenga mayor amplitud. La excepción está en el caso de que estas barras sean iguales, entonces la frecuencia real se encontraría justo en la mitad y una mínima variación de la amplitud haría que la frecuencia oscilase en 3,9 Hz. Pero éste último caso es una situación muy difícil de que se dé y en las pruebas que se han hecho no se ha detectado, por lo que se puede considerar que la frecuencia permanece estable con un error máximo de 3,9 Hz sobre la frecuencia real.

En la Figura 3.36 se muestra un ejemplo donde el resultado de la FFT no es siempre el mismo provocando una oscilación en la amplitud de las barras pertenecientes al armónico fundamental de un tono puro de 460 Hz. Este efecto se muestra en la barra de mayor amplitud donde toma valores de $2{,}85349 \cdot 10^7$ (A), $2{,}87702 \cdot 10^7$ (B), $2{,}62745 \cdot 10^7$ (C) y $2{,}79983 \cdot 10^7$ (D). Pero a pesar de esta oscilación de la amplitud ésta barra se encuentra en la misma posición en todos los casos provocando que el resultado del cálculo de la frecuencia sea siempre la mismo, y por lo tanto, estable.

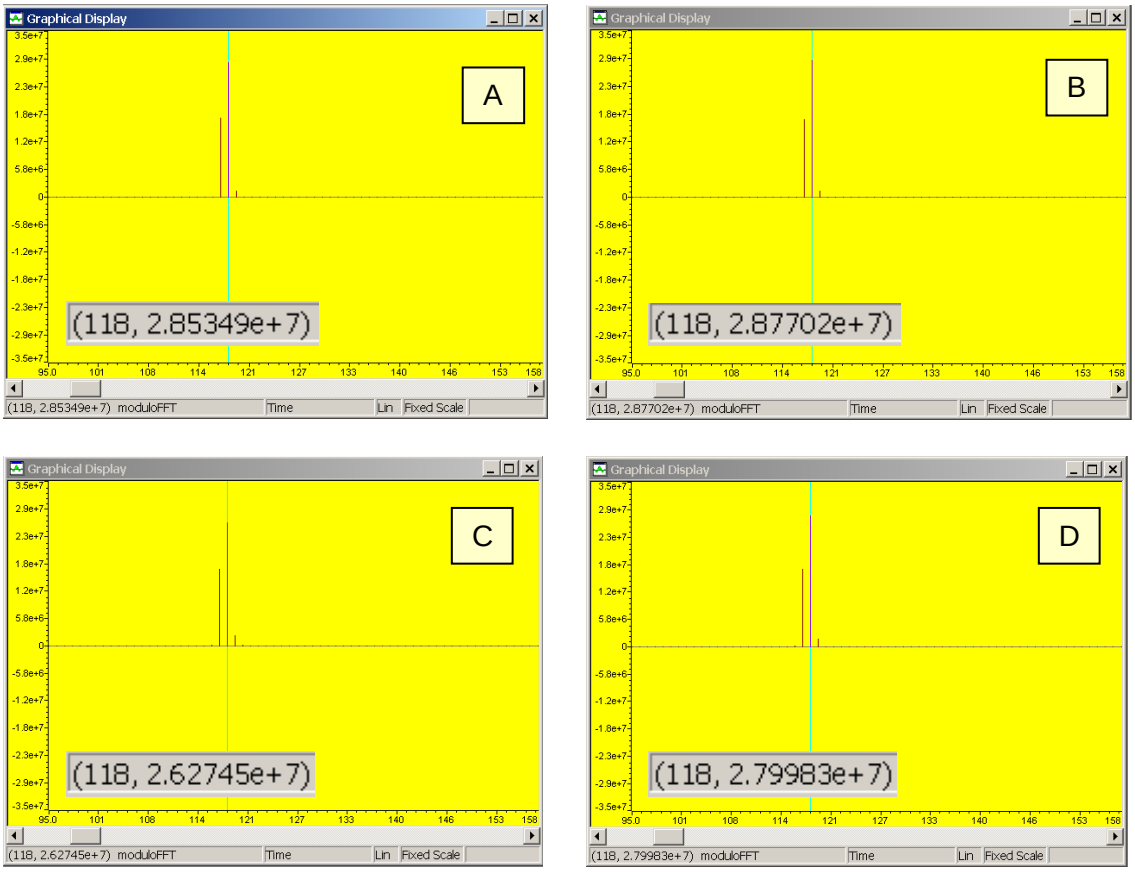


Figura 3.36 Espectro de un tono puro de 460 Hz en distintos instantes de tiempo.

Debido a esto no es necesario ni filtrar la frecuencia ni establecerla a un valor fijo por lo que se evita el retraso y se ahorra tiempo computacional.

FFT y señal de entrada

Como los datos se tienen que dividir entre el tamaño de la ventana para que no se desborde la FFT, si el valor de la amplitud de entrada es muy bajo, en el mejor de los casos puede que la amplitud máxima del espectro sea cero y por consiguiente la señal de salida también. Pero también puede ocurrir que la relación señal/ruido sea muy pequeña y el resultado de la FFT sea incorrecto debido a la proximidad con señales de ruido u otros armónicos de la propia señal reproduciendo en el peor de los casos otra señal diferente.

Los datos de la señal de entrada después de hacer la conversión analógica-digital pueden alcanzar valores de ±32768 siendo este un valor próximo a la saturación. Después de realizar varias pruebas, el cálculo de la FFT resulta correcto en señales con valores máximos alrededor de ±14000 como la de la Figura 3.37 correspondiente a un tono puro de 1 voltio de pico (Vp) de amplitud. Con ese valor máximo la señal no se satura y después de dividir por el tamaño de la ventana quedan valores máximos de ±13,6 que suponen un margen dinámico de 29 dB suficiente para calcular correctamente la FFT.

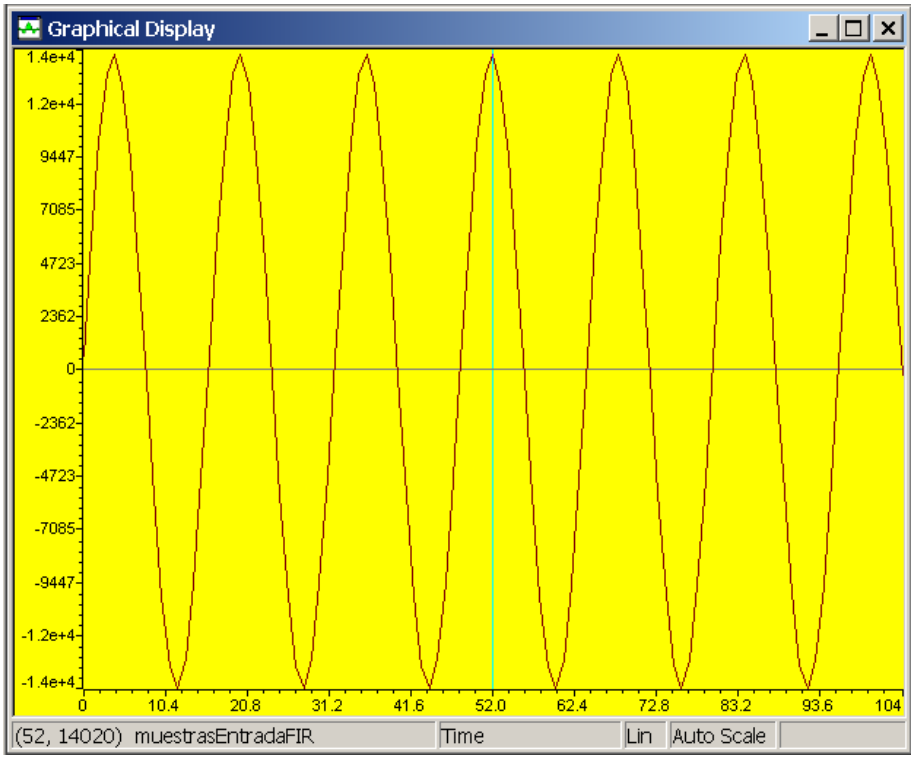


Figura 3.37 Señal de entrada perteneciente a un tono puro de 500 Hz.

En cambio en la Figura 3.38 muestra una señal de unos 0,2 Vp correspondientes a una guitarra sin amplificar cuyos valores de entrada no son mayores a ±350. Esto quiere decir que al dividir la señal por el tamaño de la ventana los valores máximos serían de ±0,34 pasando a ser cero porque la variable es entera de tipo *short*. Por lo tanto, es como si todos los valores de entrada fueran cero y como consecuencia la salida también sería cero.

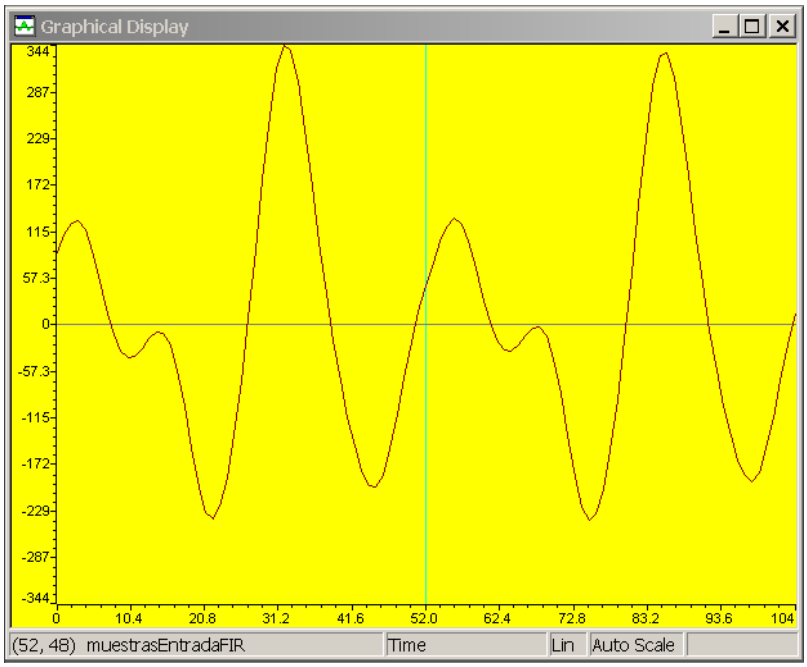


Figura 3.38 Señal de entrada perteneciente a la nota Re_3 de 146,8 Hz de una guitarra sin amplificar.

Otro caso es el de la Figura 3.39 que corresponde a la misma señal de la Figura 3.38 pero amplificada. En este caso la señal es de unos 0,7 Vp y los valores de entrada son de ±5000. Cuando los datos se dividen por el tamaño de la ventana los valores máximos de entrada pasan ser de ±4,8 que supone un margen dinámico de 19 dB aproximadamente. Este es un caso límite que donde no siempre se calcula bien la FFT, pero en cualquier caso, la amplitud resultante es tan baja que no se puede apreciar la señal de salida.

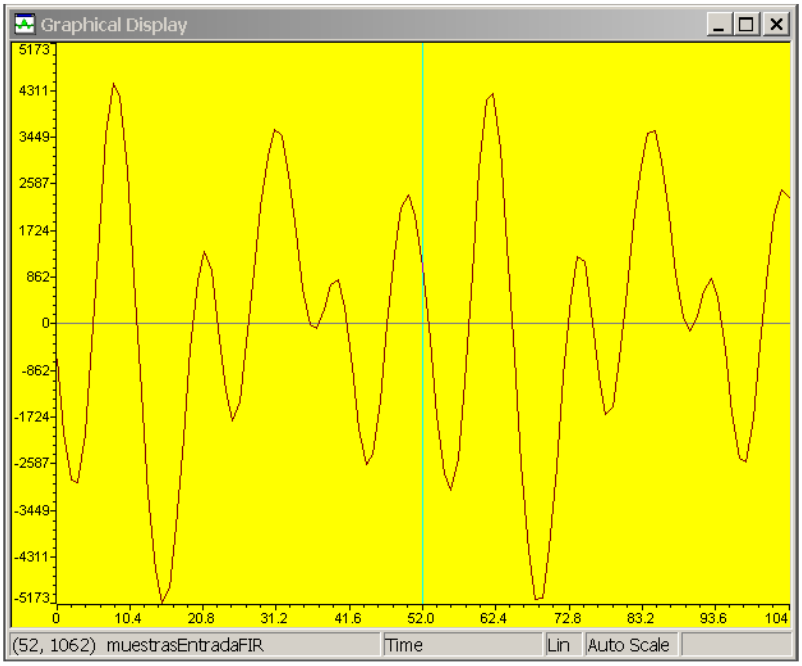


Figura 3.39 Señal de entrada perteneciente a la nota Re_3 de 146,8 Hz de una guitarra amplificada.

Teniendo en cuenta que se pueden alcanzar valores de ±32768 y que la FFT se realiza correctamente con valores de 14000, lo ideal sería que una vez divididos entre el tamaño de la ventana, los valores máximos estuvieran entre ±13,6 y ±32. En el caso de la guitarra la solución pasa por dividir los datos por un número menor al tamaño de la ventana. Así por ejemplo si los datos de ±5000 correspondientes a la señal de la guitarra amplificada, se dividen entre 250 en vez de 1024, se consiguen valores de ±20 por lo que la FFT se realizaría correctamente.

Esto significa que el número por el que se dividen los datos no debe ser fijo a 1024 y que estaría en función del tipo de señal de entrada siendo un parámetro configurable y teniendo presente que en el mejor de los casos los datos no pueden ser mayor a ±32, ya que si fueran superiores, la FFT se podría desbordar. Esto supone que la relación señal/ruido no podrá ser mayor de 36 dB.

Influencia de los armónicos

Al igual que en simulación, se han detectado casos en que cuando la señal de entrada pertenece a un instrumento como la guitarra, el segundo armónico tiene más amplitud que el fundamental, ya sea continuamente a lo largo del tiempo o sólo en algún momento determinado. En la Figura 3.40 se puede ver como el segundo armónico, incluso el tercero, tienen más amplitud que el primero o fundamental y en la Figura 3.41 es el cuarto armónico el que tiene mayor amplitud.

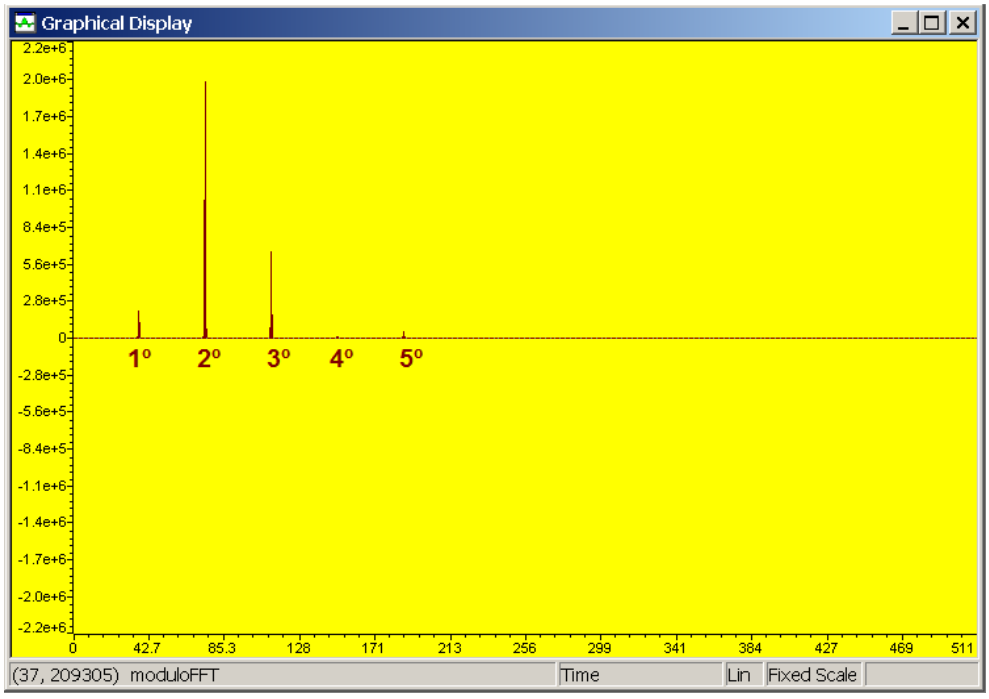


Figura 3.40 Influencia del 2º y 3er armónico.

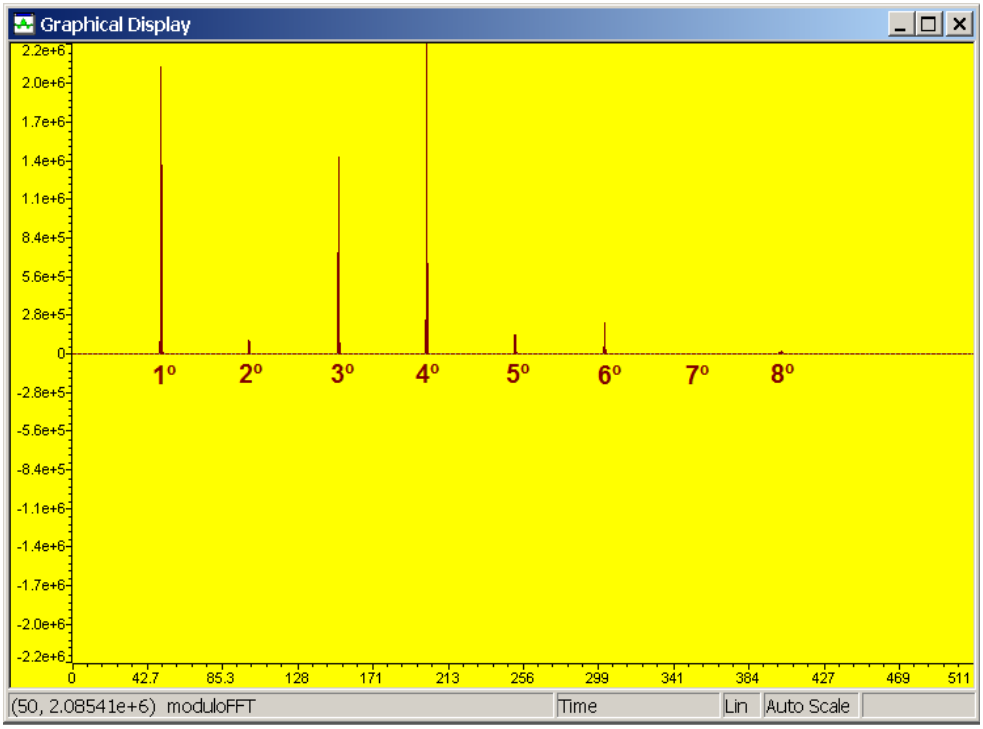


Figura 3.41 Influencia del 4º armónico.

Esto hace que el algoritmo de detección no sea robusto y que se estén detectando otras frecuencias, o que la frecuencia vaya cambiando a lo largo del tiempo para una misma nota ya que la amplitud de los armónicos no es siempre igual en todas las fases del ADSR. Por lo tanto la influencia de 2º, 3er incluso del 4º armónico implica la necesidad de mejorar el algoritmo.

Para solucionar este problema se debe minimiza la influencia de los armónicos y potenciar la frecuencia fundamental. Para ello, el algoritmo de detección se ha cambiado basándose en el algoritmo HPS (Harmonic Product Spectral). Este algoritmo hace una compresión espectral y suma de armónicos teniendo en cuenta la propiedad de que las frecuencias de los armónicos son múltiples de la frecuencia fundamental. Entonces, si se divide todo el espectro entre 2 y se multiplica por el original, la amplitud de la fundamental se suma con la del segundo armónico. Si se divide todo el espectro entre 3 y se multiplicar por el original, la amplitud de la fundamental se suma con la del tercer armónico y así sucesivamente. El resto de señales o armónicos como se multiplican por valores pequeños no son un inconveniente.

Una forma de implementarlo sería como indica la ecuación 3.21 donde se potencia la amplitud con el segundo y tercer armónico.

$$FFT[i] = FFT[i] * FFT[2i] + FFT[i] * FFT[3i] \tag{3.21}$$

Pero aplicar la ecuación 3.21 implica que el segundo y tercer armónico se encuentre exactamente en el doble y triple del índice i y esto no siempre ocurre. Incluso modificando la ecuación 3.21 por la 3.22 para coger más posiciones, pudiera darse el

caso de que tampoco se encontrara el armónico, ya que por ejemplo, si i =10, se estarían mirando las posiciones 18, 20 y 22 para el segundo armónico y 27, 30 y 33 para el tercer armónico, por lo que si la amplitud del segundo o tercer armónico estuvieran en las posiciones 17, 19, 21, 23 para el segundo o 28, 26, 29, 31, 32, 34 para el tercero, la frecuencia fundamental no sería amplificada.

$$FFT[i] = FFT[i] * FFT[2(i-1)] + FFT[i] * FFT[2i] + FFT[i] * FFT[2(i+1)] + \\ + FFT[i] * FFT[3(i-1)] + FFT[i] * FFT[3i] + FFT[i] * FFT[3(i+1)] \quad (3.22)$$

Además hay que tener en cuenta que al haber una resolución de 3,9 Hz, pueden aparecer más barras por cada armónico como muestra la Figura 3.42. Esto significa que pudiera darse el caso de que la barra más grande del armónico fundamental se amplifique con la más pequeña del segundo armónico, y que la barra más pequeña del armónico fundamental se amplificase con la más grande del segundo armónico. Esto implica que la resolución se dobla pasando a ser 7,8 Hz en el mejor de los casos, por lo que a frecuencias bajas se podría estar detectando otra nota.

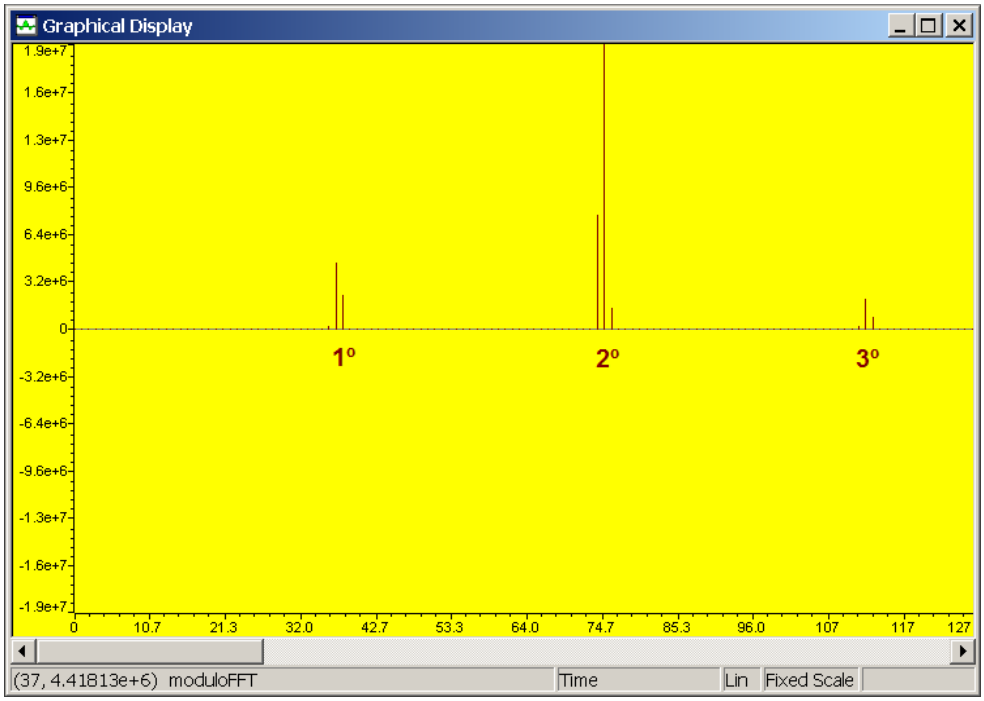


Figura 3.42 Armónicos más significativos de una nota de guitarra.

Otro inconveniente más es que recorrer todo el espectro aplicando el algoritmo HPS supone hacer muchas operaciones y aumentar el tiempo computacional de forma innecesaria ya que sólo interesa aplicarlo a la frecuencia fundamental.

La solución pasa por un proceso con diferentes pasos que se detallan a continuación.

Primero (Figura 3.42), detectar las *n* posiciones con mayor amplitud de todo el espectro y de forma ordenadas por amplitud de mayor a menor de manera que sólo se aplicará el algoritmo a esas posiciones, reduciendo así el tiempo de cálculo. Se ha

comprobado que por cada armónico pueden aparecer hasta 3 o 4 barras significativas y que pueden haber 1 o 2 armónicos mayor al fundamental por lo que se determina un valor de *n*=10. Si se cogen menos podría no cogerse la amplitud del fundamental y si se cogen más se aumenta innecesariamente el tiempo de cálculo.

Segundo (Figura 3.43), aumentar la amplitud de la diez barras detectas igualándolas a la más alta para cada armónico de forma que se asegura que la barra del armónico fundamental se amplifique con las barras más altas del segundo y tercer armónico.

Tercero (Figura 3.43), añadir por cada barra cuatro barras más, dos por delante y dos por detrás, para asegurar que al coger el doble de la posición fundamental exista el segundo o tercer armónico.

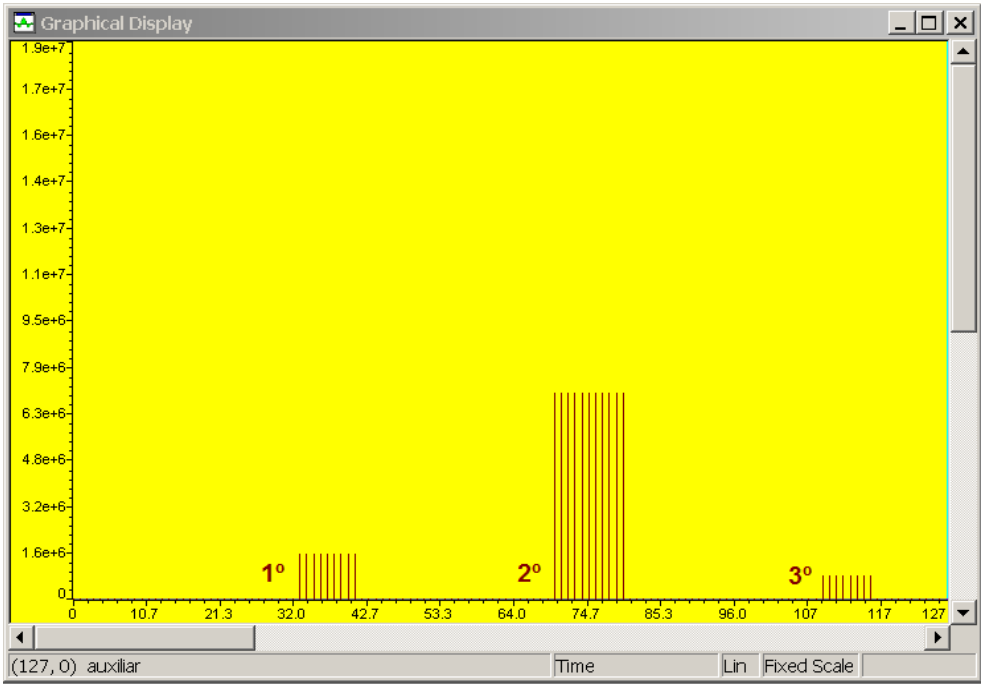


Figura 3.43 Igualación de amplitudes y creación de barras contiguas.

Cuarto (Figura 3.44), amplificar el valor de cada posición utilizando la ecuación 3.23, pero sólo de las 10 barras detectadas y no de las que se han creado artificialmente. Como se puede observar la ecuación no es exactamente igual a la 3.21 ya que con sólo sumar las amplitudes ya es suficiente y sólo se aplica a las posiciones detectadas con mayor amplitud. También se puede ver en la Figura 3.44 que el segundo y tercer armónico no se han amplificado ya que los valores situados en el doble o triple de su posición son insignificantes.

$$FFT[i_n] = FFT[i_n] + FFT[2i_n] + FFT[3i_n] \tag{3.23}$$

Quinto, se recorren las 10 posiciones para detectar el valor máximo. Como se han igualado amplitudes la posición corresponde a la detectada anteriormente en primer lugar.

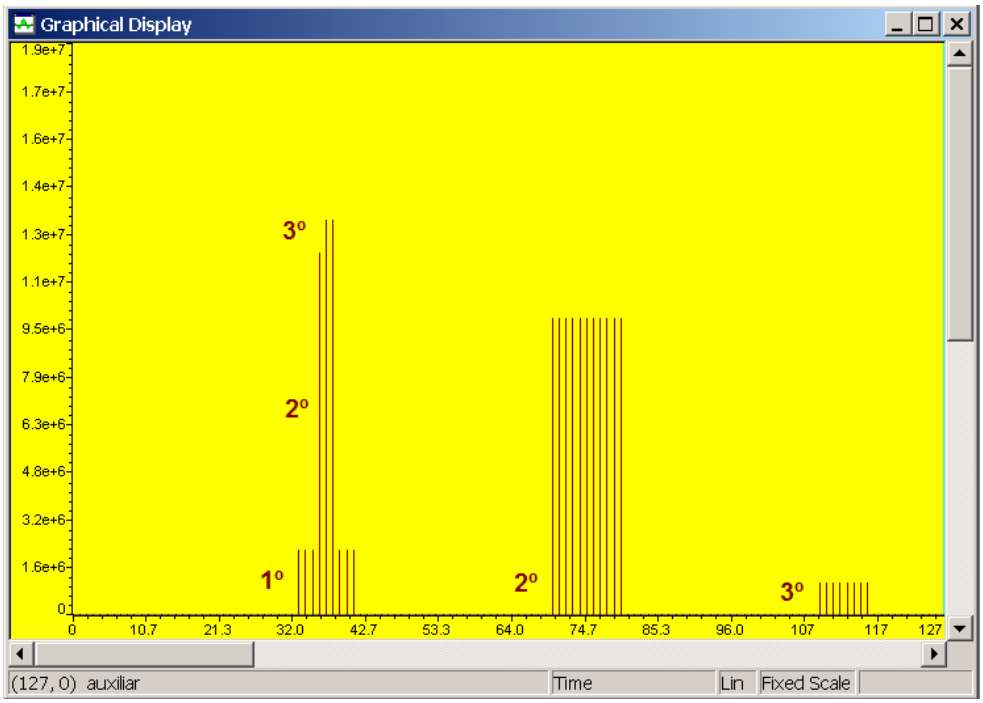


Figura 3.44 Suma del 2º y 3er armónico al fundamental.

Finalmente, con la posición detectada correspondiente al primer armónico ya se puede determinar la amplitud tan solo mirando el valor que hay en el espectro para esa posición, y calcular la frecuencia fundamental usando la ecuación 3.5.

En la Figura 3.45 se muestra el diagrama de flujo y en la Figura 3.46 el algoritmo mejorado que detecta la frecuencia fundamental y su amplitud.

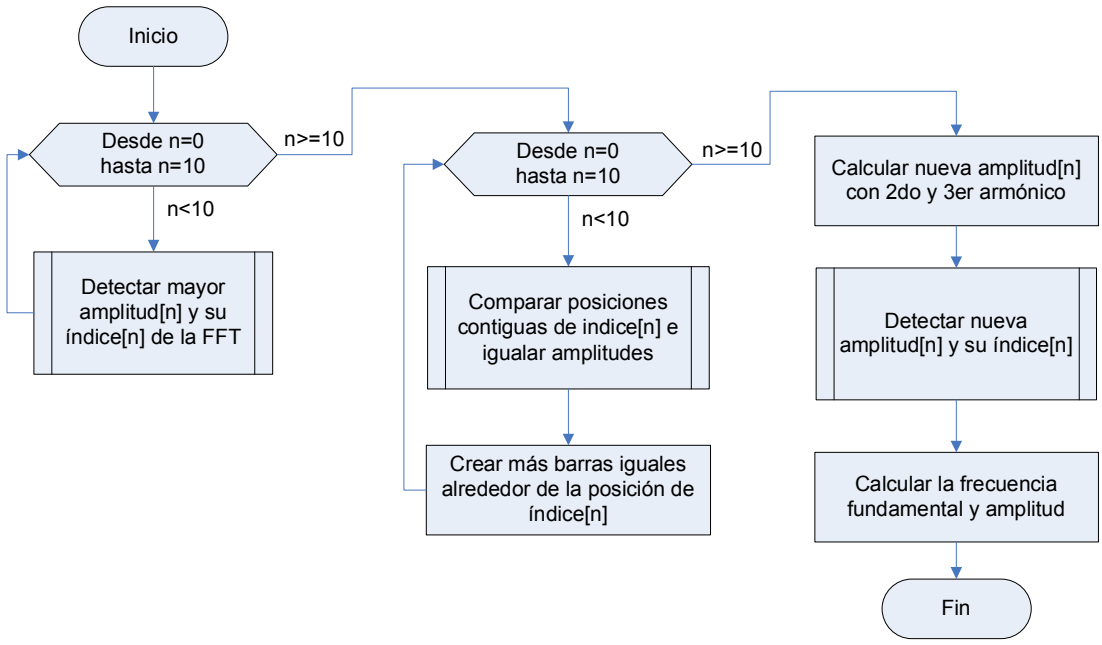


Figura 3.45 Diagrama de flujo del algoritmo mejorado de detección de la frecuencia fundamental y su amplitud.

```
// Seleccionar los índices de las amplitudes mas grandes:
for(i=0; i<NUM_IND; i++){
    amplitudes[i] = 0;
    for (j=iMin; j<iMax; j++) {
        if (moduloFFT[j]>amplitudes[i]) {
            amplitudes[i] = moduloFFT[j];
            indices[i] = j;
        }
    }
    moduloFFT[indices[i]] = 0;
}
// Igualar amplitudes de posiciones contiguas:
for(i=0; i<NUM_IND; i++){
    moduloFFT[indices[i]] = amplitudes[i]; // Restaurar amplitudes
    for(j=0; j<NUM_IND; j++){
         if(indices[i]+1==indices[j] || indices[i]-1==indices[j]
           || indices[i]+2==indices[j]|| indices[i]-2==indices[j]){
              if(amplitudes[i]<amplitudes[j]){
                  amplitudes[i]=amplitudes[j];
              }
         }
     // Colocar las amplitudes de los índices elegidos en el vector
     // auxiliar:
     auxiliar[indices[i]-3] = amplitudes[i];
     auxiliar[indices[i]-2] = amplitudes[i];
     auxiliar[indices[i]-1] = amplitudes[i];
     auxiliar[indices[i]] = amplitudes[i];
     auxiliar[indices[i]+1] = amplitudes[i];
     auxiliar[indices[i]+2] = amplitudes[i];
     auxiliar[indices[i]+3] = amplitudes[i];
}
// Amplificar amplitud fundamental con 2do y 3er armónico:
for (i=0; i<NUM_IND; i++){
    auxiliar[indices[i]] = auxiliar[indices[i]] +
                           auxiliar[indices[i]*2] +
                           auxiliar[indices[i]*3];
}
// Eliminar valores fuera del rango:
auxiliar[iMin-3] = 0;
auxiliar[iMin-2] = 0;
auxiliar[iMin-1] = 0;
auxiliar[iMax+1] = 0;
auxiliar[iMax+2] = 0;
auxiliar[iMax+3] = 0;
// Buscar la amplitud máxima:
amplitudEspectralMax = 0;
indiceAmplitudEspectralMax = 0;
for (i=0; i<(NUM_IND); i++) {
    if (auxiliar[indices[i]]>amplitudEspectralMax) {
        amplitudEspectralMax = auxiliar[indices[i]];
        indiceAmplitudEspectralMax = indices[i];
    }
}
// Calcular amplitud y frecuencia fundamental:
amplitudEspectralMax =
sqrt((double)moduloFFT[indiceAmplitudEspectralMax]);
frecuenciaFundamental = (float)indiceAmplitudEspectralMax *
                        resolucionFrecuencia;
```

Figura 3.46 Algoritmo mejorado en CCS de detección de la frecuencia fundamental y su amplitud.

3.2.5 Efectos

En la Tabla 3.5 se muestran los efectos de armonía que puede reproducir esta aplicación que tienen como base teórica la explicada en el capítulo 2. La diferencia entre uno y otro está básicamente en el número de notas que se van a reproducir y la frecuencia de cada una de ellas. Esta frecuencia se determina en base a la frecuencia fundamental detectada (f_0) de forma general y en el caso de los acordes teniendo en cuenta también la escala y la tonalidad.

En la Tabla 3.5 se muestra las frecuencias que le corresponden a cada nota según el efecto. En algunos efectos se pude calcular de forma directa y en otros casos se utiliza la ecuación 2.1 para determinar la frecuencia de la nota donde f_0 es la frecuencia de referencia, S la raíz doceava de dos correspondiente a un semitono y el número de semitonos es siempre siete para el caso de las quintas y para los acordes se deja en función de la escala y la tonalidad, donde A es el intervalo de la tercera, B es intervalo de la quinta y C es el intervalo de la séptima.

Efecto	Nº de notas	Frecuencia 1ª nota	Frecuencia 2ª nota	Frecuencia 3ª nota	Frecuencia 4ª nota
Bypass	1	f_0	-	-	-
Octavador 1	2	f_0	$2 \cdot f_0$	-	-
Octavador 2	3	f_0	$2 \cdot f_0$	$f_0/2$	-
Quintas	2	f_0	$f_0 \cdot S^7$	-	-
Quintas + octava	3	f_0	$f_0 \cdot S^7$	$2 \cdot f_0$	-
Acorde tríada	3	f_0	$f_0 \cdot S^A$	$f_0 \cdot S^B$	-
Acorde cuatríada	4	f_0	$f_0 \cdot S^A$	$f_0 \cdot S^B$	$f_0 \cdot S^C$
Diseño de armonía	de 2 a 4	f_0	$f_0 \cdot S^a$	$f_0 \cdot S^b$	$f_0 \cdot S^c$

Tabla 3.5 Frecuencias de las notas según efecto.

En el diseño de armonía se crean varios intervalos armónicos donde se debe indicar el número de intervalos y la distancia en semitonos *a*, *b* y *c* respecto la nota detectada. Como no se trata de un acorde, no se tiene en cuenta ni la escala ni la tonalidad.

Acordes

Para calcular las frecuencias de un acorde se utiliza el mismo planteamiento y algoritmo que en simulación pero usando como nota más baja la Do_2 y no la Mi_2, ya que la resolución frecuencial lo permite y de esta forma se amplía el rango de utilización de instrumentos. Esto supone que la frecuencia de comparación (f_{comp}) sea 63,57090194 Hz. Por lo tanto, lo que primero se hace es llevar la frecuencia detectada

a la más baja (f_{Low}) de la segunda octava y después determinar el número de notas que hay entre ésta frecuencia y la más baja (Do_2). Este número de notas corresponde la posición de una tabla donde se indica el número de intervalos para formar el acorde.

Lo que ocurre es que en simulación sólo se habían tabulado los intervalos para la escala mayor natural en la tonalidad de *Do* y ahora se tiene que tener en cuenta el resto de escalas y de tonalidades para formar el acorde. Revisando la Tabla 2.4 se puede ver que hay 9 escalas diferentes y que cada una de ellas tiene 15 tonalidades por lo que harían falta 135 tablas para tabular todos los intervalos posibles.

Hay que tener en cuenta que todas las escalas, excepto la menor armónica y la menor melódica, están relacionadas con la escala mayor natural ya que tienen la misma estructura y los mismos intervalos para formar acordes pero empezando en grados diferentes. Esto quiere decir que, para una misma tonalidad, se puede encontrar el acorde de una nota de otra escala modificando la posición en la tabla de la escala mayor natural.

En la Tabla 3.6 se han tabulado los intervalos para las notas de la escala mayor natural y del modo dórico. Como se puede observar para cada posición de la tabla los intervalos puede ser diferentes en cada escala, pero si se suman dos posiciones en la escala mayor natural se consiguen exactamente los intervalos para formar los acorde en el modo dórico. También se podrían restar 10 posiciones ya que como la secuencia de notas se va repitiendo se puede considerar el desplazamiento la tabla circular. Por ejemplo, para el modo dórico, el acorde de la nota *Do* está en la posición 2, el acorde de la nota *Fa* está en la posición 7 y el acorde de *Si* está en la posición 1 en la tabla de la escala mayor natural.

Posición		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Nota		**Do_2**	**$Do\#_2$ Reb_2**	**Re_2**	**$Re\#_2$ Mib_2**	**Mi_2**	**Fa_2**	**$Fa\#_2$ $Solb_2$**	**Sol_2**	**$Sol\#_2$ Lab_2**	**La_2**	**$La\#_2$ Sib_2**	**Si_2**
Escala mayor natural	**Estruc.**		T		T	S		T		T		T	S
	A	4	0	3	0	3	4	0	4	0	3	0	3
	B	7	0	7	0	7	7	0	7	0	7	0	6
	C	11	0	10	0	10	11	0	10	0	10	0	10
Modo dórico	**Estruc.**		T		S	T		T		T		S	T
	A	3	0	3	4	0	4	0	3	0	3	4	0
	B	7	0	7	7	0	7	0	7	0	6	7	0
	C	10	0	10	11	0	10	0	10	0	10	11	0

Tabla 3.6 Relación de intervalos con la escala mayor natural.

Esto se puede hacer con el resto de escalas obteniendo así el modificador a emplear en cada caso. En el caso de las escalas menor armónica y menor melódica como son escalas artificiales se deben tratar en tablas aparte por lo que no tienen modificador.

Además de seleccionar la escala también se puede seleccionar la tonalidad. También existe una relación entre tonalidades por lo que si se quiere tocar en otra tonalidad partiendo de la tonalidad de *Do*, lo único que hay que hacer es modificar la posición de la nota que marca la tonalidad y llevarla a la posición de la nota que marca la tonalidad de *Do* que es la 0 en la tabla. Así por ejemplo, como se puede ver en la Tabla 3.7 sólo basta con restar 2 posiciones o sumarle 10 para poder formar acordes en la tonalidad de *Re* usando la tabla de la tonalidad de *Do*.

Posición	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Tonalidad de Do	Do	Do#	Re	Re#	Mi	Fa	Fa#	Sol	Sol#	La	La#	Si
Tonalidad de Re	Re	Re#	Mi	Fa	Fa#	Sol	Sol#	La	La#	Si	Do	Do#
A	4	0	3	0	3	4	0	4	0	3	0	3
B	7	0	7	0	7	7	0	7	0	7	0	6
C	11	0	10	0	10	11	0	10	0	10	0	10

Tabla 3.7 Relación de intervalos con la tonalidad de *Do*.

Esto da lugar a otro modificador, por lo que al final se deberá aplicar sobre la posición de la tabla obtenida inicialmente los modificadores por escala y tonalidad para obtener los intervalos definitivos para formar un acorde. En la Tabla 3.8 y en la Tabla 3.9 se muestran estos modificadores.

Escala / Modo	**Modificador**
Mayor natural / Jónico	+0
Dórico	+2
Frigio	+4
Lidio	+5
Mixolidio	+7
Menor natural / Eólico	+9
Locrio	+11
Menor armónica	+0
Menor melódica	+0

Tabla 3.8 Modificadores de la posición de la tabla según la escala o modo.

Tonalidad	Modificador
Do/Si#	+0
Do#/Reb	+11
Re	+10
Re#/Mib	+9
Mi/Fab	+8
Fa/ Mi#	+7
Fa#/Solb	+6
Sol	+5
Sol#/Lab	+4
La	+3
La#/Sib	+2
Si/Dob	+1

Tabla 3.9 Modificadores de la posición de la tabla según la tonalidad.

Entonces para determinar el acorde, se necesitan dos variables, relacionadas con la escala y la tonalidad y sólo tres tablas en vez de 135: una tabla exclusiva para formar los acordes de la escala menor armónica (Tabla 3.11), otra tabla exclusiva para los acordes de la escala menor melódica (Tabla 3.12) y una tabla, basada en la escala mayor para formar los acordes del resto de escalas (Tabla 3.10). Las tres tablas parten de la tonalidad de *Do.*

Posición	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Nota	**Do_2**	**$Do\#_2$**	**Re_2**	**$Re\#_2$**	**Mi_2**	**Fa_2**	**$Fa\#_2$**	**Sol_2**	**$Sol\#_2$**	**La_2**	**$La\#_2$**	**Si_2**
A	4	0	3	0	3	4	0	4	0	3	0	3
B	7	0	7	0	7	7	0	7	0	7	0	6
C	11	0	10	0	10	11	0	10	0	10	0	10

Tabla 3.10 Intervalos para formar acordes partiendo de la escala mayor natural.

Posición	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Nota	**Do_2**	**$Do\#_2$**	**Re_2**	**$Re\#_2$**	**Mi_2**	**Fa_2**	**$Fa\#_2$**	**Sol_2**	**$Sol\#_2$**	**La_2**	**$La\#_2$**	**Si_2**
A	3	0	3	4	0	3	0	4	4	0	0	3
B	7	0	6	8	0	7	0	7	7	0	0	6
C	11	0	10	11	0	10	0	10	11	0	0	9

Tabla 3.11 Intervalos para formar acordes de la escala menor armónica.

Posición	1	2	3	4	5	6	7	8	9	10	11	12
Nota	Do_2	$Do\#_2$	Re_2	$Re\#_2$	Mi_2	Fa_2	$Fa\#_2$	Sol_2	$Sol\#_2$	La_2	$La\#_2$	Si_2
A	3	0	3	4	0	3	0	4	0	3	0	3
B	7	0	7	8	0	7	0	7	0	6	0	6
C	11	0	10	11	0	10	0	10	0	10	0	10

Tabla 3.12 Intervalos para formar acordes de la escala menor melódica.

3.2.6 Generar la señal de salida

La señal de salida correspondiente a una nota está definida por la ecuación 3.24

$$salidaNota[k] = A_0 \cdot \sin\left(2\pi \cdot f_{Nota} \cdot t\right) \tag{3.24}$$

Como para una misma nota la frecuencia es la misma, la señal se repite periódicamente donde el tiempo t indica porque parte del periodo se encuentra la señal. Esto quiere decir que teniendo los valores correspondientes a un periodo ya es suficiente para reproducir la señal de forma indefinida. Por lo tanto, para reconstruir una señal se necesita generar una tabla con los valores de un período, entre 0 y 2π, de la función seno donde la frecuencia natural de la tabla sea menor o igual a la frecuencia mínima que se desea reconstruir. La longitud de la tabla depende de esta frecuencia y viene dada por la ecuación 3.25.

$$long.tabla \geq \frac{Fs}{frec.mínima} \tag{3.25}$$

Si la *Fs* son 8kHz y la frecuencia mínima es de 65,4 Hz, la longitud mínima de la tabla es de 123 valores. Pero cuantos más valores tenga la tabla más calidad tendrá la señal por lo que se aumenta a 8000 valores de manera que la frecuencia mínima sería de 1 Hz siendo ésta la frecuencia natural de la tabla (f_{tabla}). La tabla se llena con la función seno correspondiente a un periodo de la señal mediante la ecuación 3.26.

$$tabla[i] = sin \cdot \frac{2\pi \cdot i}{long.tabla} \tag{3.26}$$

Donde 0≤ i <$long.tabla$ y los datos de la tabla corresponden al valor de la señal en un período de frecuencia de 1 Hz. En la Figura 3.47 se puede ver la representación de esta señal hecha con valores tipo *short* para poderlos usar en el programa.

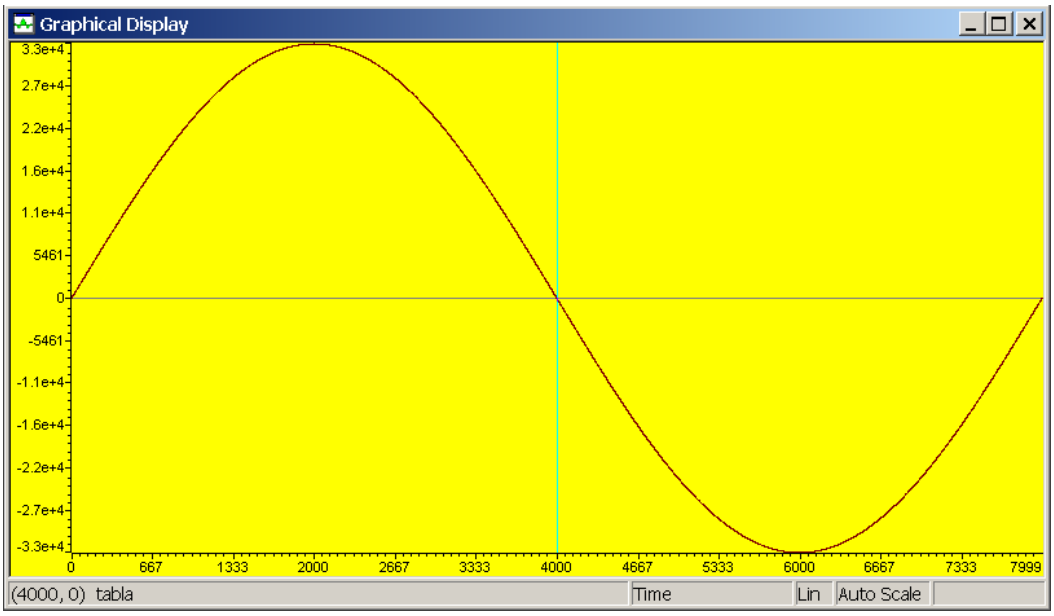


Figura 3.47 Señal base para reconstruir las señales de salida

A partir de esta señal se puede reconstruir cualquier otra de frecuencia igual o superior a 1 Hz cogiendo un número de valores determinados de la tabla. Al principio, para la primera muestra de la señal de salida se coge el valor de la tabla correspondiente a la primera posición. Para la siguiente muestra de salida se cogerá el valor de la siguiente posición de la tabla según las ecuaciones 3.27 y 3.28 y así sucesivamente hasta que se llega al final de la tabla donde continúa por el principio como si de un buffer circular se tratase.

$$posición[k] = posición\,[k-1] + incremeto \tag{3.27}$$

$$Incremento = \frac{f_{Nota}}{f_{tabla}} \tag{3.28}$$

Esto quiere decir que para reconstruir una señal de 1 Hz hay que coger para cada muestra de salida los valores de la tabla con incrementos de una posición, por lo que la señal se repite cada 8000 muestras. En cambio, si se quiere reconstruir una señal de 2 Hz, para cada muestra de la señal de salida se deben coger los valores de la tabla cada dos posiciones por lo que se necesitan 4000 muestras para reproducir un periodo entero. Entonces Cuanto mayor sea la frecuencia menos valores se utilizan, ya que el incremento es más grande, y se necesitan menos muestras para completar un período. Por ejemplo para reproducir una señal de 1kHz, se cogen los valores de la tabla cada 1000 posiciones y se necesitan 8 muestras para completar un periodo.

Esto quiere decir que la ecuación de la señal de salida de una nota viene redefinida por la ecuación 3.29.

$$salidaNota[k] = A_0 \cdot \text{tabla}[posición[k]] \tag{3.29}$$

Donde $tabla[posición[k]] = \sin(2\pi \cdot f_{Nota} \cdot t)$

En el caso de una armonía, donde hay más de una nota, se usará la misma tabla pero con incrementos diferentes para cada señal siendo la señal de salida final la expresada por la ecuación 3.30 donde 0≤n<4

$$salida[k] = \sum_{i=0}^{n} salidaNota_i[k] \quad (3.30)$$

El inconveniente de sumar varías señales es que se podría saturar la señal de salida por eso conviene tener un parámetro configurable que permita atenuarla y controlar la saturación. Esto se consigue mediante una variable que multiplica el valor de la amplitud detectada por un valor inferior a 1. También puede ser interesante poder aumentar la señal de salida en el caso de que ésta sea pequeña por lo que la variable además podría funcionar a modo de volumen. En esta aplicación se ha utilizado para esta variable un rango de 0,1 < *variableSalida* < 2.

Respecto a la cantidad de datos de salida, como la *Fs* de salida también es 8 kHz, el número de muestras de salida también son igual al paso. En la Figura 3.48 se muestra el diagrama de flujo y en la Figura 3.49 el algoritmo que calcula la señal de salida en función del número de notas.

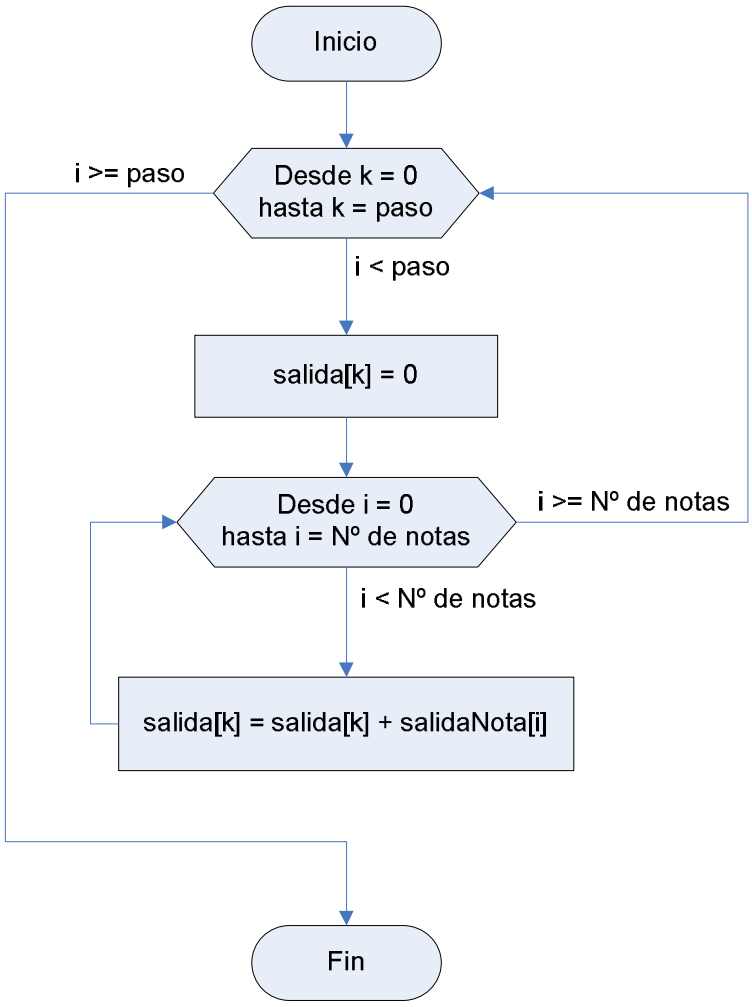


Figura 3.48 Diagrama de flujo del algoritmo que calcula la señal de salida

```
// Generar incrementos para cada frecuencia:
for(i=0; i<numNotas; i++){
    incrementoTabla[i] = f[i]/frecNaturalTabla;
}

// Generar señal de salida:
for (i=0; i<PASO; i++){
    salida[i] = 0;
    for(j=0; j<numNotas; j++){
        salida[i] = salida[i] + (amplitudEspectralMax * tabla[indT[j]]);
        // Actualizar índices:
        indiceTabla[j] = indiceTabla[j] + incrementoTabla[j];
        if (indiceTabla[j] >= LONG_TABLA){
            indiceTabla[j] = indiceTabla[j] - LONG_TABLA;
        }
        indT[j] = indiceTabla[j]; // Coge la parte entera
    }
    salida[i] = salida[i]>>15; // Se pasar a short
}
```

Figura 3.49 Algoritmo en CCS que calcula la señal de salida.

Salida de datos

La salida de datos también se realiza mediante la técnica del doble buffer. Como la transmisión de datos en el McBSP puede ser simultánea, al mismo tiempo que se están adquiriendo los datos del buffer de entrada se sacan los datos del buffer de salida en el mismo ciclo de trabajo mientras se procesan los datos del otro buffer de entrada En la Tabla 3.13 se muestran las operaciones actualizadas que realizan el EDMA y la CPU en cada ciclo.

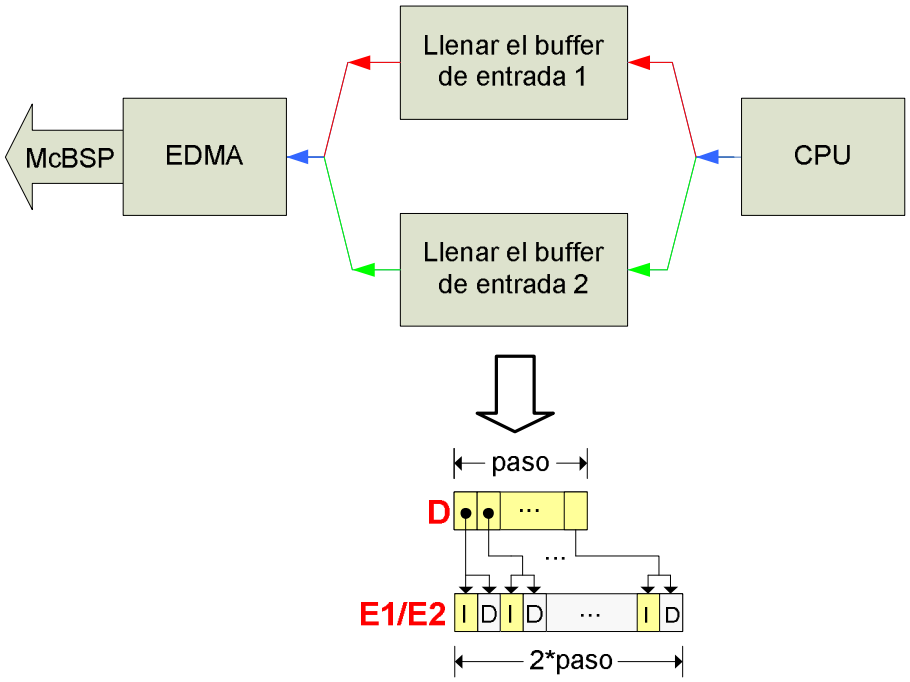
Ciclo	EDMA	CPU
0	Adquisición de datos en bufferIn 1 Salida de datos en bufferOut 1	Procesar datos del bufferIn 2
1	Adquisición de datos en bufferIn 2 Salida de datos en bufferOut 2	Procesar datos del bufferIn 1

Tabla 3.13 Operaciones actualizadas que realizan el EDMA y la CPU en cada ciclo.

La *Fs* del códec de salida es igual que la de entrada antes de decimar, es decir de 8 kHz, por lo que se necesitan la misma cantidad de muestras tanto en la adquisición como en la salida de datos por lo que los buffers de salida son de la misma longitud que los de entrada, esto es igual al doble del paso.

Por lo tanto el paso es el mismo tiempo en que se están sacando/adquiriendo muestras.

Como la señal de salida que se genera no es estéreo y para no dejar un canal a cero, cada muestra de salida se copia en los dos canales. En la Figura 3.50 se pude ver como se realiza la transferencia de datos en la salida.



D ⇒ Espacio de memoria reservado para la señal de salida
E1/E2 ⇒ Espacios de memoria reservados para los buffers de salida

Figura 3.50 Transferencia con el EDMA en la salida de datos.

Señales de salida

A continuación se muestran varias señales de salida correspondientes a diferentes efectos generados a partir de un tono puro. En la Figura 3.51 pertenece a un bypass formado por una única señal por lo que tiene la misma forma que un tono puro.

Para el resto de figuras la señal de salida es el resultado de la suma de 2, 3 o 4 señales. La forma no es siempre la misma y depende de cómo se encuentren desfasadas las señales entre sí en cada instante de tiempo. El octavador es una excepción y mantiene siempre la misma forma ya que al ser la señal de la octava superior o inferior múltiples de la fundamental no existe desfase o éste es siempre el mismo.

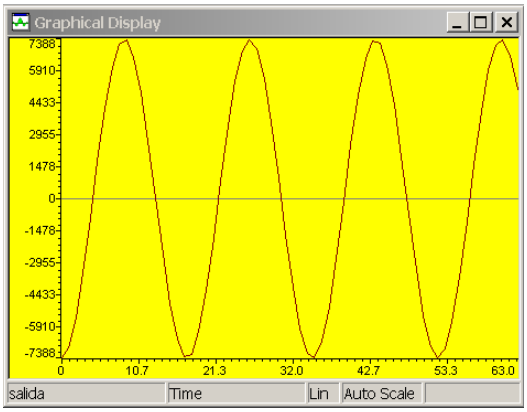


Figura 3.51 Bypass

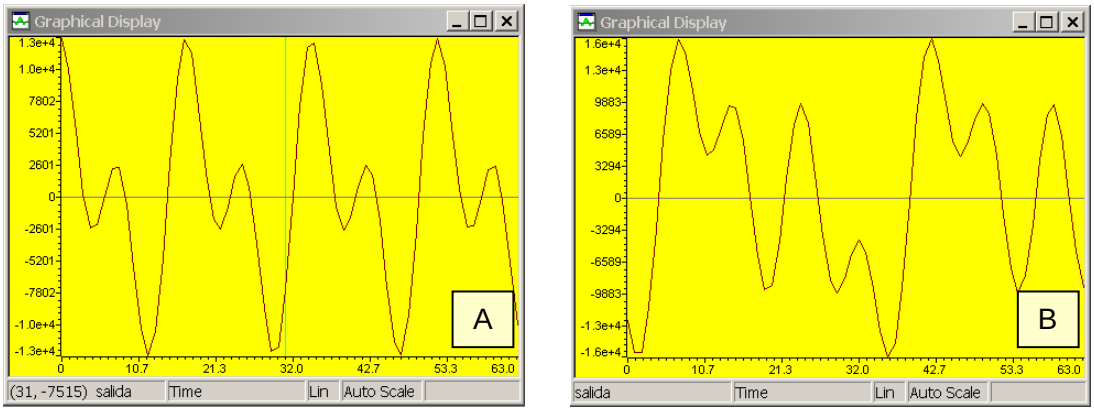


Figura 3.52 Octavador. (A) Con octava superior. (B) Con octava superior e inferior.

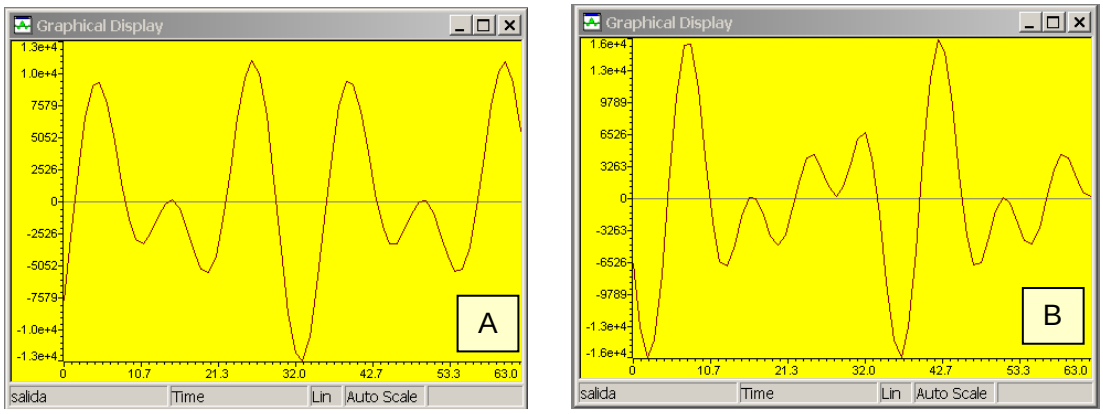


Figura 3.53 Quintas. (A) Intervalo de quintas. (B) Intervalo de quintas con octava.

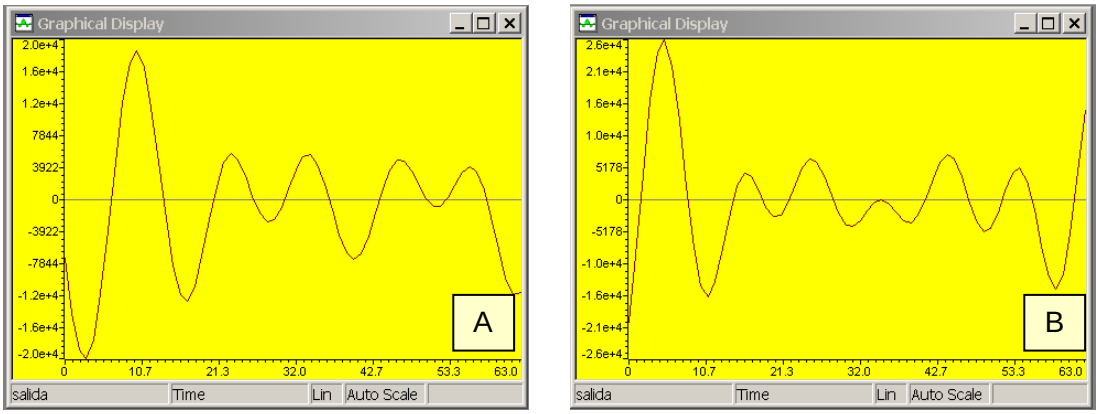


Figura 3.54 Acordes. (A) Tríada. (B) Cuatríada.

3.2.7 Resultados de la implementación con un DSP

En la Tabla 3.14 están los resultados obtenidos después de realizar pruebas con una guitarra eléctrica. Para cada nota de la tabla se indica el valor de la frecuencia teórica y el de la obtenida experimentalmente. El valor teórico corresponde al calculado con la ecuación 2.1 cogiendo como frecuencia de referencia el La_4 a 440 Hz y el valor experimental a la frecuencia fundamental obtenida con el DSP. También se indica en la tabla la diferencia de frecuencias entre el valor teórico y el experimental y el margen de frecuencia que hay con la nota siguiente.

Como se puede observar, los resultados no coinciden con el valor teórico aunque se aproxima bastante. Esto es debido a la resolución que tiene la aplicación y a la afinación del instrumento.

Esta diferencia entre la frecuencia teórica y la detectada experimentalmente varía entre 0,5 Hz y 5 Hz. Esto quiere decir que ha habido casos en que se ha superado los 3,9 Hz de resolución, pero esta variación es aceptable si se tiene en cuenta que entre dos notas consecutivas hay una margen de frecuencia, y que éste es mayor cuanto más grande sea la frecuencia de la nota.

Para el caso más desfavorable de la guitarra la nota más baja es la Mi_2, donde la variación entre esta nota y la siguiente es de 4,9 Hz. Para esta nota, se ha obtenido una variación de 0,4 Hz por lo que se estaría dentro del margen.

En los casos que la diferencia de frecuencia ha superado a la resolución no se ha superado el margen, siendo estos:

- La nota Do_4 de 261,6 Hz con una diferencia de 4 Hz pero un margen de 15,5 Hz.
- La nota Mi_5 de 659,2 Hz con una diferencia de 4,8 Hz pero con un margen de 39,2 Hz.
- La nota Fa_3 de 174,3 Hz con una diferencia de 5 Hz pero con un margen de 10,3 Hz.

Incluso cuando la diferencia ha sido entre 3 y 4 Hz el margen es de 9 a 15 Hz y para las frecuencias bajas con un margen más ajustado la diferencia ha sido de 0,3 a 0,7 Hz con márgenes de 4,9 a 6,9 Hz. En el caso de haber un silencio la frecuencia se mantiene a cero.

Con la mejora de algoritmo de detección de la frecuencia fundamental, no se ha dado ningún caso en que la frecuencia detectada fuera la de otro armónico.

Notas	Frec. Teórica (Hz)	Frec. Experimental (Hz)	Δf (Hz)	Margen con siguiente nota (Hz)
Do_3	130,8	128,9	1,9	7,7
$Do\#_3$	138,5	136,7	1,8	8,3
Do_4	261,6	265,6	4	15,5
Re_3	146,8	144,5	2,3	8,7
$Re\#_3$	155,5	152,3	3,2	9,3
Re_4	293,6	296,8	3,2	17,5
Mi_2	82,4	82	0,4	4,9
Mi_3	164,8	164	0,8	9,8
Mi_4	329,6	328,1	1,5	19,6
Mi_5	659,2	664	4,8	39,2
Fa_3	174,6	179,6	5	10,3
$Fa\#_3$	184,9	187,5	2,6	11
Fa_4	349,2	351,5	2,3	22
Sol_2	97,9	97,6	0,3	5,9
Sol_3	195,9	199,2	3,3	11,7
$Sol\#_3$	207,6	210,9	3,3	12,4
Sol_4	391,9	390,6	1,3	23,4
La_2	110	109,3	0,7	6,5
$La\#_2$	116,5	117,1	0,6	6,9
La_3	220	222,6	2,6	13
$La\#_3$	233	234,3	1,3	13,9
Si_2	123,4	121	2,4	7,4
Si_3	246,9	250	3,1	14,7
Silencio	0	0	0	-

Tabla 3.14 Resultados la aplicación con DSP.

3.3 Interfaz gráfica

Esta aplicación tienes varios parámetros que son configurables. En función del valor de la variable asociada a cada uno de ellos se puede:

- Seleccionar el efecto.
- Cambiar de escala.
- Cambiar de tonalidad.
- Aumentar la ganancia de entrada
- Controlar la amplitud de salida.
- Diseñar una armonía.

Además existe un parámetro más, que no se ha comentado hasta ahora pero que se explica más adelante, que permite que suene o no una nota que no se encuentra dentro de una escala.

Cada vez que se quieren cambiar los valores de estos parámetros hay que parar el programa, modificar los valores en el CCS, volver a compilar y ejecutarlo de nuevo. Esto a nivel de usuario es un inconveniente por lo que se decide hacer un entorno gráfico de forma que desde el PC se puedan realizar las actualizaciones de las variables sin detener la ejecución del programa.

En la Figura 3.55 se muestra la pantalla que se ha desarrollado con Visual C++ (VC++) para configurar la aplicación.

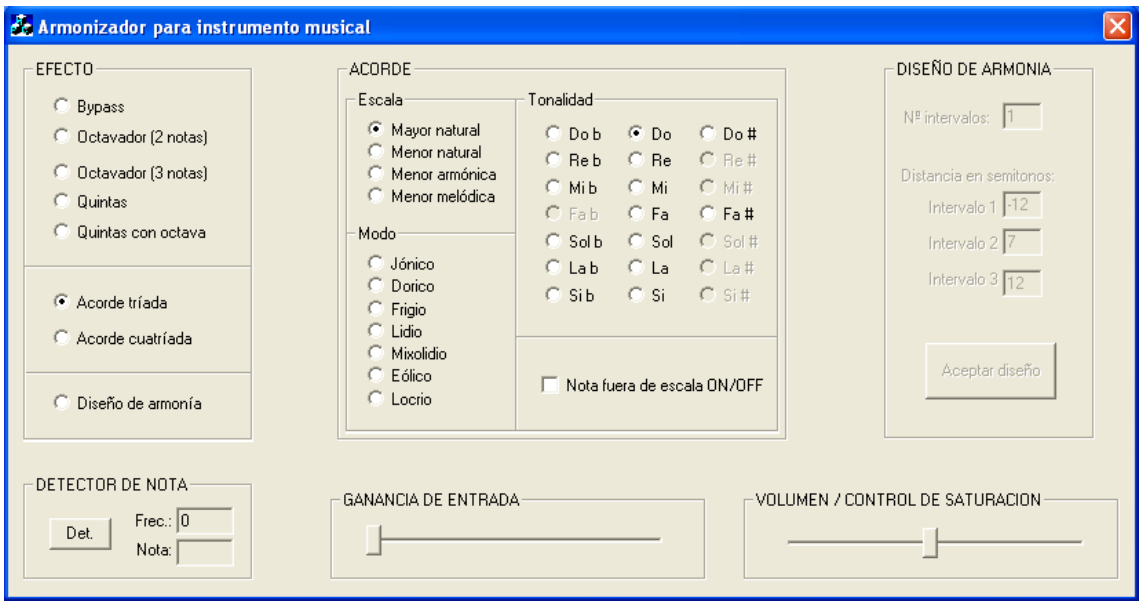


Figura 3.55 Ventana para configurar la aplicación.

En esta aplicación sólo se transfieren datos en un sentido, desde el PC al DSP, pero se deja el programa preparado para comunicarse en ambos sentidos para futuras ampliaciones de la aplicación. Por ejemplo, el bloque de la pantalla denominado

DETECTOR DE NOTA, es una mejora futura y pretende mostrar la frecuencia y la nota que se está tocando por lo que la información procede del DSP.

Por lo tanto, cada vez que se actúa sobre la ventana se envía el dato al DSP para que pueda procesarlo. La comunicación se realiza mediante el protocolo RTDX que permite el envío de datos en tiempo real.

En la parte de VC++ no se realizan operaciones ni se procesan datos, sólo se envían datos en función del estado de la ventana: estado de botones, posición del cursor en las barras de desplazamiento, estado de la casilla de verificación, valores de los cuadros de edición, etc. Después, el DSP, con los valores que le llegan realiza las acciones y cálculos oportunos. De esta forma se hace una parte independiente de la otra.

En el apartado de la implementación con VC++ se explica que hace cada componente de la ventana y el valor que se envía al DSP cuando recibe un evento. En la implementación con CCS se explica que se hace con los valores que le llegan desde el PC.

3.3.1 Comunicación RTDX

RTDX (Real-Time Data Exchange) es un protocolo que permite el intercambio de datos en tiempo real entre el PC (Host) y el DSP (Target) mediante la interfaz JTAG (Joint Test Action Group) al mismo tiempo que el procesador está ejecutando la aplicación.

Estos datos se pueden tratar desde el PC mediante una automatización OLE (Object Linking and Embedding), que es una forma que tiene Windows para comunicarse entre programas y que se puede programar con VC++.

Para el intercambio de información se usan unas rutinas que ya vienen definidas en las librerías del RTDX.

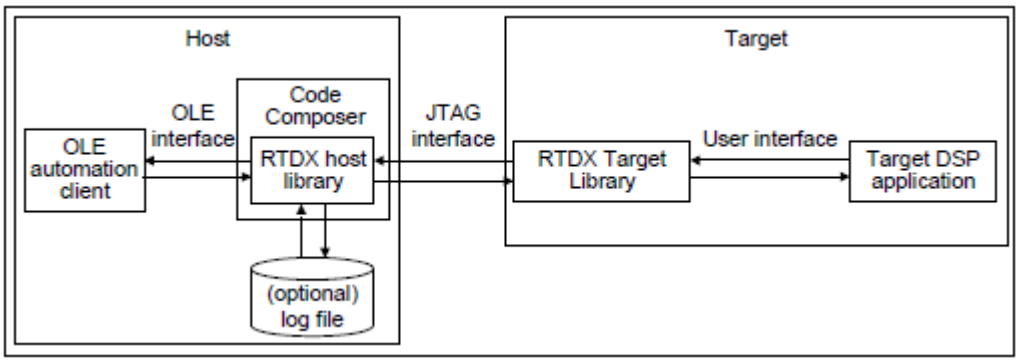


Figura 3.56 Flujo de datos RTDX entre el PC (Host) y el DSP (Target).

Trama de datos en la comunicación entre el PC y el DSP

Para la transferencia de datos se usa una array con valores enteros. Para esta aplicación se necesitan 10 posiciones donde en función del valor de cada posición se determinan y configuran los efectos y se modifica la ganancia de entrada y la amplitud de salida.

La array en VC++ se llama *rtdxOutputData[]* porque es de salida y en el CCS se llama *rtdxInputData[]* porque es de entrada. En la Tabla 3.15 se muestran los valores que puede tener cada posición del array para cada parámetro configurable de esta aplicación, y como se inicializan. Más adelante en la implementación con VC++ y CCS se explica con más detalle su significado.

Posición del array	Parámetro	Rango	Valor inicial
0	Efecto	0 – 7	0
1	Escala	0 – 10	0
2	Tonalidad	0 – 11	0
3	Nota fuera de escala	0 – 1	0
4	Ganancia de entrada	0 – 100	1
5	Volumen	0 – 100	48
6	Nº de intervalos	1 – 3	1
7	Intervalo *a*	(-12) – 12	-12
8	Intervalo *b*	(-12) – 12	7
9	Intervalo *c*	(-12) – 12	12

Tabla 3.15 Relación de las posiciones del array de comunicación con los parámetros configurables de la aplicación y su rango de valores.

3.3.2 Implementación con Visual C++

Para que el PC pueda comunicarse con el DSP hay que crear la clase *IRtdxExp* en VC++. El CCS incorpora el fichero *Rtdxint.dll* que proporciona a VC++ una serie de funciones necesarias para el intercambio de datos y que ya están implementadas. También hay que añadir la función *AfxOleInit()* dentro la función *InitInstance()* en la clase que define los comportamientos de la aplicación, para habilitar la comunicación de objetos OLE.

En la Figura 3.57 se muestra el código relacionado con la comunicación que se ejecuta cuando se inicializa la aplicación. En él se crean los objetos *w_RTDX* para enviar datos y *r_RTDX.*para recibir datos, necesarios para poder usar las funciones de clase *IRtdxExp*.

```
// Escribir datos
w_RTDX = new IRtdxExp;
w_RTDX->CreateDispatch(_T("RTDX"));
if(!w_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
    MessageBox("Imposible inicializar el procesador","Error");

// Leer datos
r_RTDX = new IRtdxExp;
r_RTDX->CreateDispatch(_T("RTDX"));
if(!r_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
    MessageBox("Imposible inicializar el procesador","Error");

// Configuración del RTDX para la recepción de datos:
// Modo continuo,4 buffers de 1024
w_RTDX->DisableRtdx();
w_RTDX->ConfigureRtdx(1,1024,4);
w_RTDX->EnableRtdx();

// Mira si el canal de escritura (W) está abierto
if(w_RTDX->Open("rtdxInputChannel","W"))
     MessageBox("No se puede abrir el canal","Error");

// Mira si el canal de lectura (R) está abierto
if(r_RTDX->Open("rtdxOutputChannel","R"))
     MessageBox("No se puede abrir el canal","Error");
```

Figura 3.57 Inicialización de la comunicación en VC++.

A pesar de que esta aplicación sólo envía datos desde el PC a la DSP, están implementadas las funciones en VC++ que permiten la comunicación en los dos sentidos. Para enviar datos desde VC++ al DSP se utiliza la función *OnEnviarDatos()* y su implementación se muestra en la Figura 3.58.

```
void CArmonizadorDlg::OnEnviarDatos()
{
    VARIANT sa;                  // Declaración del objeto
    SAFEARRAYBOUND rgsabound[1]; // Dimensión del objeto
    ::VariantInit(&sa);          // Inicialización del objeto
    sa.vt = VT_ARRAY | VT_I4;    // Objeto de enteros de 4 bytes
    rgsabound[0].lLbound = 0;    // Valor mínimo del incremento
    rgsabound[0].cElements = LONG_RTDX; // Nº elementos igual al
                                        // tamaño del array de
                                        // comunicaciones
    // Se crea el objeto con la configuración anterior:
    sa.parray = SafeArrayCreate(VT_I4, 1, rgsabound);
    // Se pasan al objeto los valores del array de comunicación:
    HRESULT hr;
    for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++)
    {
        hr = ::SafeArrayPutElement(sa.parray, &i,
                                   (long*)&rtdxOutputData[i]);
    }
    long bufferstate;
    w_RTDX->Write(sa, &bufferstate); //Se envía el objeto al buffer
    ::VariantClear(&sa); // Se limpia el valor del objeto
    w_RTDX->Flush();     // Se envían los datos al DSP
}
```

Figura 3.58 Implementación en VC++ de la función para enviar datos al DSP.

Para recibir datos en VC++ desde el DSP se utiliza la función *OnRecibirDatos()* y su implementación se muestra en la Figura 3.59.

```
void CArmonizadorDlg::OnRecibirDatos()
{
    VARIANT sa;                  // Declaración del objeto
    SAFEARRAYBOUND rgsabound[1]; // Dimensión del objeto
    ::VariantInit(&sa);          // Inicialización del objeto
    sa.vt = VT_ARRAY | VT_R4;    // Objeto de floats de 4 bytes
    rgsabound[0].lLbound = 0;    // Valor mínimo del incremento
    rgsabound[0].cElements = LONG_RTDX; // Nº elementos igual al
                                        // tamaño del array de
                                        // comunicaciones
    // Se crea el objeto con la configuración anterior:
    sa.parray = SafeArrayCreate(VT_R4, 1, rgsabound);
    r_RTDX->ReadSAF4(&sa);
    r_RTDX->Rewind();
    // Se pasan a la array de comunicaciones los valores del objeto:
    HRESULT hr;
    for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++)
    {
        hr = ::SafeArrayGetElement(sa.parray, &i,
                                   &rtdxInputData[i]);
    }
    ::VariantClear(&sa); // Se limpia el valor del objeto
}
```

Figura 3.59 Implementación en VC++ de la función para recibir datos desde el DSP.

Rutinas de ventana

Se han creado diferentes funciones para controlar el aspecto y manipulación de la ventana. Se utilizan para habilitar y deshabilitar los componentes de la ventana en función de las opciones seleccionadas. Para ello se usa la función *EnableWindow()* pasándole el parámetro TRUE para habilitar o FALSE para deshabilitar.

Como el efecto seleccionado por defecto es el bypass, y todos los componentes están habilitados, la ventana se inicializa desactivando todos los componentes de los bloques *ACORDE* y *DISEÑO DE ARMONIA* mediante las funciones *OnDesactivarAcorde()* y *OnDesactivarArmonia()* respectivamente.

Los componentes de los bloques de *EFECTOS*, *GANANCIA DE ENTRADA* y *VOLUMEN/CONTROL DE SATURACIÓN* están siempre activos.

En la Tabla 3.16 se muestran las funciones que controlan la ventana y sobre que componentes actúan.

Función VC++	Acción
OnDesactivarAcorde()	Desactiva los botones de opción de escala/modo, tonalidad y la casilla de nota fuera de escala.
OnActivarAcorde()	Activa los botones de opción de escala/modo, tonalidad y casilla de nota fuera de escala.
OnDesactivarArmonia()	Desactiva el botón y los cuadros de edición del diseño de armonía.
OnDesactivarTonosMayor()	Desactiva los botones de opción de las tonalidades no disponibles en la escala mayor natural y modo jónico.
OnDesactivarTonosMenor()	Desactiva los botones de opción de las tonalidades no disponibles en la escala menor natural y modo eólico.
OnDesactivarTonosDorico()	Desactiva los botones de opción de las tonalidades no disponibles en el modo dórico.
OnDesactivarTonosFrigio()	Desactiva los botones de opción de las tonalidades no disponibles en el modo frigio.
OnDesactivarTonosLidio()	Desactiva los botones de opción de las tonalidades no disponibles en el modo lidio.
OnDesactivarTonosMixolidio()	Desactiva los botones de opción de las tonalidades no disponibles en el modo mixolidio.
OnDesactivarTonosLocrio()	Desactiva los botones de opción de las tonalidades no disponibles en el modo locrio.
OnActivarTonosEscala()	Activa los botones de opción de las tonalidades en función de la escala seleccionada.
OnDesactivarTonos()	Desactiva todos los botones de opción de las tonalidades.
OnActivarTonos()	Activa los botones de opción de todas las tonalidades.
OnIniciarTono()	Selecciona el botón de opción de la Tonalidad de Do.
OnActivarArmonia()	Activa el botón y los cuadros de edición del diseño de armonía para un intervalo.
OnActualizarArmonia()	Activa los cuadros de edición del diseño de armonía en función del número de intervalos.

Tabla 3.16 Funciones para controlar la ventana.

Opciones efecto

En la Figura 3.60 se puede ver el bloque de la ventana que permite elegir un efecto. Cada vez que se selecciona un botón se deselecciona el anterior por lo que sólo puede haber seleccionado un efecto a la vez.

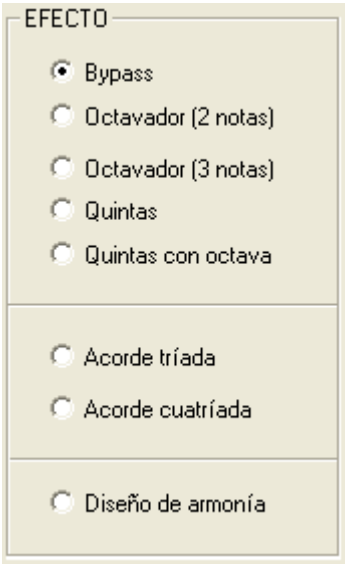


Figura 3.60 Bloque de efectos.

Cuando se selecciona un botón:

- Se activan los componentes de los bloques de la ventana relacionados con ese efecto.
- Se desactivan los componentes de los bloques de la ventana que no están relacionados con ese efecto.
- Se actualiza el contenido de *rtdxOutputData[0]* según la Tabla 3.17.
- Se envían los datos al DSP. El dato que se envía sirve para identificar el tipo de efecto.

Función VC++	Efecto	rtdxOutputData[0]
OnEfBypass()	Bypass	0
OnEfOctavador1()	Octavador 1 octava	1
OnEfOctavador2()	Octavador 2 octavas	2
OnEfQuintas1()	Quintas	3
OnEfQuintas2()	Quintas con octava	4
OnEfTriada()	Acorde tríada	5
OnEfCuatriada()	Acorde cuatríada	6
OnEfArmonia()	Diseño de armonía	7

Tabla 3.17 Valor de *rtdxOutputData[0]* según el efecto.

Opciones escala

En la Figura 3.61 se puede ver el bloque de la ventana que permite elegir una escala o un modo. Cada vez que se selecciona un botón se deselecciona el anterior por lo que sólo puede haber seleccionado una escala o un modo.

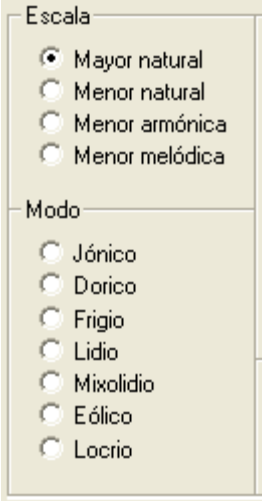


Figura 3.61 Bloque de escala/modo

Cuando se selecciona un botón:

- Se activan todos los componentes de las tonalidades.
- Se desactivan los componentes de las tonalidades que no relacionados con esa escala o modo.
- Se selecciona por defecto la tonalidad de *Do* por lo que *rtdxOutputData[2]* =0.
- Se actualiza el contenido de *rtdxOutputData[1]* según la Tabla 3.18.
- Se envían los datos al DSP. El dato que se envía sirve para identificar el tipo de escala o modo iniciándose en la tonalidad de *Do*.

Función VC++	Escala/modo	rtdxOutputData[1]
OnEscMayor()	Mayor	0
OnEscMenorNatural()	Menor natural	1
OnEscMenorArmonica()	Menor armónica	2
OnEscMenorMelodica()	Menor melódica	3
OnEscJonico()	Modo jónico	4
OnEscDorico()	Modo dórico	5
OnEscFrigio()	Modo frigio	6
OnEscLidio()	Modo lidio	7
OnEscMixolidio()	Modo mixolidio	8
OnEscEolico()	Modo eólico	9
OnEscLocrio()	Modo locrio	10

Tabla 3.18 Valor de *rtdxOutputData[1]* según la escala o modo.

Opciones Tonalidad

En la Figura 3.62 se puede ver el bloque de la ventana que permite elegir una tonalidad. Cada vez que se selecciona un botón se deselecciona el anterior por lo que sólo puede haber seleccionado una tonalidad.

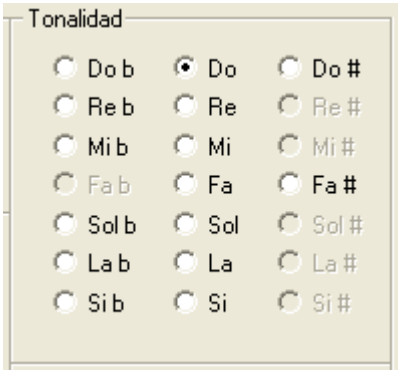


Figura 3.62 Bloque de tonalidad.

Cuando se selecciona un botón:

- Se actualiza el contenido de *rtdxOutputData[2]* según la Tabla 3.19.
- Se envían los datos al DSP. El dato que se envía sirve para identificar el tipo de tonalidad.

Función VC++	Tonalidad	rtdxOutputData[2]
OnTonoSiS() / OnTonoDo()	Si# / Do	0
OnTonoDoS() / OnTonoReB()	Do# / Reb	1
OnTonoRe()	Re	2
OnTonoReS() / OnTonoMiB()	Re# / Mib	3
OnTonoMi() / OnTonoFaB()	Mi / Fab	4
OnTonoMiS() / OnTonoFa()	Mi# / Fa	5
OnTonoFaS() / OnTonoSolB()	Fa# / Solb	6
OnTonoSol()	Sol	7
OnTonoSolS() / OnTonoLaB()	Sol# / Lab	8
OnTonoLa()	La	9
OnTonoLaS() / OnTonoSiB()	La# / Sib	10
OnTonoSi() / OnTonoDoB()	Si / Dob	11

Tabla 3.19 Valor de *rtdxOutputData[2]* según la tonalidad.

Casilla de verificación de nota fuera de escala

En la Figura 3.63 se puede ver el bloque de la ventana que permite que suene o no una nota que no se encuentra en la escala seleccionada.

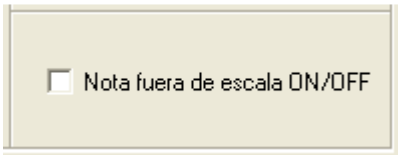


Figura 3.63 Bloque de nota fuera de escala.

La casilla de verificación tiene dos estados. Si se selecciona la casilla (ON) su estado es *TRUE* y si se deselecciona (OFF) su estado es *FALSE*. Cada vez que se actúa sobre la casilla de verificación:

- Se actualiza el contenido de *rtdxOutputData[3]* según la Tabla 3.20.
- Se envían los datos al DSP. El dato que se envía sirve para identificar si la nota fuera de escala debe sonar o no.

Función VC++	Estado	rtdxOutputData[3]
OnCheckNotaNoEscala()	FALSE	0
	TRUE	1

Tabla 3.20 Valor de *rtdxOutputData[3]* según la nota fuera de escala.

Barras de desplazamiento

En la Figura 3.64 se pueden ver los bloques de la ventana que permiten aumentar la ganancia de entrada (A) y controlar la amplitud de salida (B).

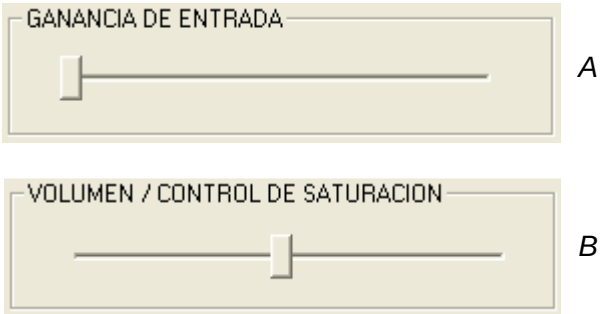


Figura 3.64 Bloques de ganancia de entrada (A) y control de la amplitud de salida (B).

En función de la posición del cursor de la barra de desplazamiento se adquieren valores entre 0 y 100. Siendo 0 cuando el cursor está en el extremo izquierdo y 100 cuando está en el extremo derecho.

Cada vez que se actúa sobre el cursor:

- Se actualiza el contenido de *rtdxOutputData[4]* o *rtdxOutputData[5]* según la posición del cursor.
- Se envían los datos al DSP. El dato que se envía sirve para aumentar la ganancia de la señal de entrada o para controlar la saturación/volumen de la amplitud de la señal de salida según el caso.

Función VC++	Array	Valor
OnReleasedcaptureSliderGainIn()	rtdxOutputData[4]	0 – 100
OnReleasedcaptureSliderVolumen()	rtdxOutputData[5]	0 – 100

Tabla 3.21 Rango de valores de *rtdxOutputData[4]* y *rtdxOutputData[5]* para las barras de desplazamiento.

Cuadros de edición y entrada de datos para el diseño de armonía

En la Figura 3.65 se puede ver el bloque de la ventana que permite diseñar una armonía. Este es el único bloque con entrada de datos numéricos por el usuario en la aplicación.

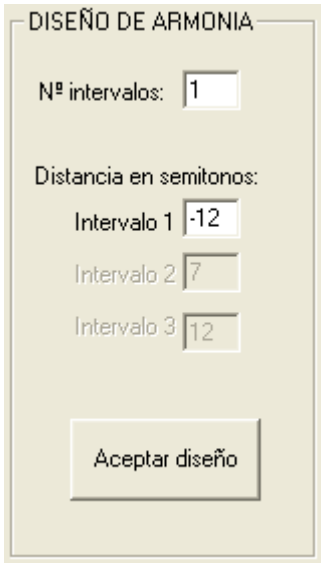


Figura 3.65 Bloque de diseño de armonía.

Cada vez que se cambia el numero de intervalos se actualiza la pantalla y se habilitan o deshabilitan los componentes de cada intervalo. En la Figura 3.66 se pueden ver las tres situaciones que se pueden dar.

El rango de valores para cada intervalo está comprendido entre -12 y 12 semitonos por lo que se puede conseguir hasta una octava por arriba o una octava por debajo de la nota detectada.

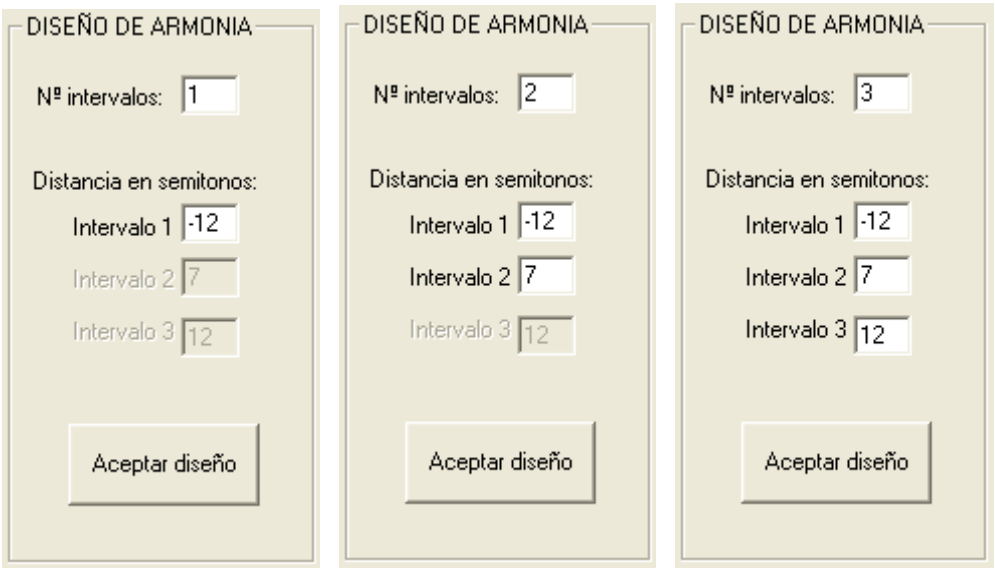


Figura 3.66 Estado del bloque de diseño de armonía en función del número de intervalos.

VC++ permite configurar el rango de valores que se pueden entrar en cada cuadro de edición de forma que se genera automáticamente una ventana con un mensaje de error si se introduce un valor incorrecto. Para que no se abandone el cuadro de edición si se ha hecho una entrada incorrecta se utilizan las funciones *OnChange()* y *OnKillfocus()* en cada cuadro de manera que lo actualizan cuando éste cambia o se abandona. De esta forma la ventana de error aparece en el momento que el dato que se introduce incorrectamente o se actúa sobre otro componente de la ventana. La Figura 3.67, la Figura 3.68 y la Figura 3.69 muestran los distintos mensajes de error para el diseño de armonía.

Figura 3.67 Mensaje de error si no se entra un número entero.

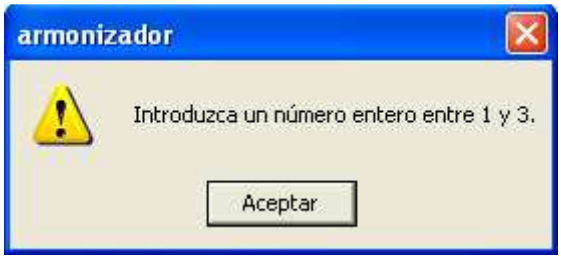


Figura 3.68 Mensaje de error si no se entra un número entre 1 y 3 para el nº de intervalos.

Figura 3.69 Mensaje de error si no se entra un número entre -12 y 12 para el nº de semitonos.

Una vez se han entrado los datos correctamente se pueden validar con el botón *Aceptar diseño*. Cada vez que se pulsa este botón:

- Se actualiza el contenido de *rtdxOutputData[6], rtdxOutputData[7], rtdxOutputData[8]* y *rtdxOutputData[5]* con el dato de los cuadros de edición.
- Se envían los datos al DSP. Los datos que se envía sirven para determinar el número de notas que tiene la armonía y calcular la frecuencia de cada una de ellas.

Función VC++	Array	Valor
OnButtonArmonia()	rtdxOutputData[6]	1 – 3
	rtdxOutputData[7]	(-12) – 12
	rtdxOutputData[8]	(-12) – 12
	rtdxOutputData[9]	(-12) – 12

Tabla 3.22 Rango de valores de *rtdxOutputData[6], rtdxOutputData[7], rtdxOutputData[8]* y *rtdxOutputData[9]* para los cuadros de edición.

3.3.3 Implementación con CCS

Para recibir datos desde el PC se debe crear en el CCS un canal de entrada con la función *RTDX_CreateInputChannel()* y para que pueda haber una transferencia de datos se debe habilitar con la función *RTDX_enableInput().* Una vez que se ha creado y habilitado el canal de entrada sólo hay que esperar a que lleguen datos al DSP.

Para recibir datos desde el PC se usan conjuntamente las funciones *RTDX_channelBusy()* y *RTDX_readNB()*. La función *RTDX_channelBusy()* mira si el canal de entrada está libre o está siendo usado. Si el canal está ocupado la función no se realiza ninguna lectura, pero si el canal está libre se ejecuta la función *RTDX_readNB().* Esta función hace una solicitud de lectura y espera a que lleguen datos al canal para leerlos. La ventaja que tiene esta función es que no bloquea el programa mientras no llegan datos por lo que el resto la aplicación continúa ejecutándose normalmente mientras tanto. En el momento que lleguen datos estos pasan a la array.

En el CCS, el programa va mirando para cada vuelta del bucle infinito si llegan datos desde el PC. En el momento que se reciben datos se actualiza array de comunicación de entrada *rtdxInputData[]*. En la Figura 3.70 se puede ver el diagrama de flujo de la aplicación DSP incluyendo la parte de comunicación y en la Figura 3.71 el código en para la lectura de datos.

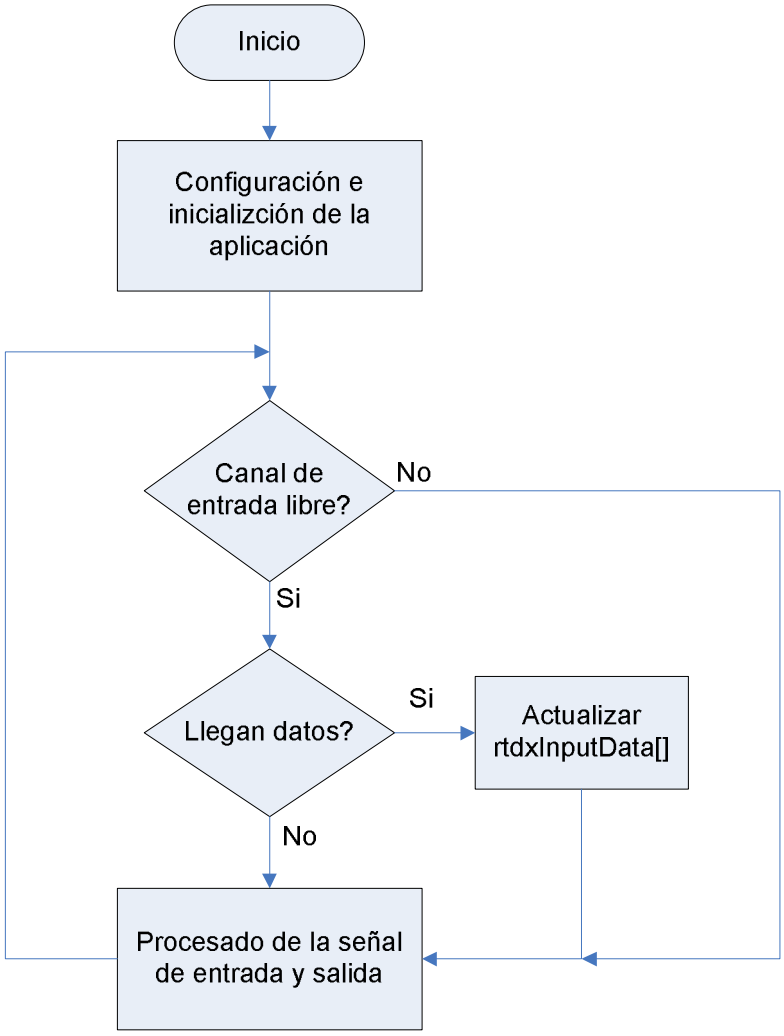


Figura 3.70 Diagrama de flujo de la aplicación DSP con lectura de datos.

```
//Mira si el canal esta libre
if(!RTDX_channelBusy(&rtdxInputChannel)){

// Lectura de datos. Se queda esperando a que le
// lleguen datos pero el código se sigue ejecutando
    RTDX_readNB(&rtdxInputChannel, &rtdxInputData,
        sizeof(rtdxInputData));
}
```

Figura 3.71 Código en CCS para recibir datos desde el PC.

A pesar de que en esta aplicación sólo existe la comunicación en un sentido (del PC al DSP) el programa está preparado para realizar la comunicación en el otro sentido (del DSP al PC) para futuras mejoras o actualizaciones.

Para que pueda haber una transferencia de datos debe existir un canal pero no se puede usar un mismo canal para recibir y enviar datos por lo que deben crearse por separado. Para enviar datos desde la DSP se debe crear en el CCS un canal de salida con la función *RTDX_CreateOutputChannel ()* y para que puede haber una transferencia de datos se debe habilitar con la función y *RTDX_enableOutput()*.

Para enviar datos se usa la función RTDX_write() la cual coloca los datos en el buffer de la tarjeta RTDX. En la Figura 3.72 se muestra el código para la escritura de datos.

```
// Escritura de datos
RTDX_write(&rtdxOutputChannel,
           &rtdxOutputData,
           sizeof(rtdxOutputData));
```

Figura 3.72 Código en CCS para escribir datos en el PC.

rtdxInputData[0]

En función del valor de *rtdxInputData[0]* se determina el número de notas y se calcula la frecuencia de cada una de ellas en base a la Tabla 3.5. Se utiliza una estructura *switch-case* tal y como muestra la Figura 3.73.

```
switch (rtdxInputData[0]){
    case 0: // Bypass
        f[0] = frecuenciaFundamental;
        numNotas = 1;
        break;
    case 1: // Octavador 1
        f[0] = frecuenciaFundamental;
        f[1] = frecuenciaFundamental*2;
        numNotas = 2;
        break;
    case 2: // Octavador 2
        f[0] = frecuenciaFundamental;
        f[1] = frecuenciaFundamental*2;
        f[2] = frecuenciaFundamental/2;
        numNotas = 3;
        break;
    case 3: // Quintas 1
        f[0] = frecuenciaFundamental;
        f[1] = frecuenciaFundamental * pow(SEMITONO, 7);
        numNotas = 2;
        break;
    …
    …
```

Figura 3.73 Código en CCS para determinar el efecto y calcular las frecuencias de cada nota.

rtdxInputData[1]

En función del valor de *rtdxInputData[1]* se determina el modificador por escala en base a la Tabla 3.8. También se determina la tabla de intervalos que se va a usar para formar el acorde en base a la Tabla 3.10, la Tabla 3.11 y la Tabla 3.12. Se utiliza una estructura *switch-case* tal y como muestra la Figura 3.74.

```
if(rtdxInputData[1]!=escala){ // Si no cambia la escala no se hace
                              //el switch
    escala = rtdxInputData[1];
    switch (rtdxInputData[1]){
        case 0: modificadorEscala = MAYOR; break;
        case 1: modificadorEscala = MENOR_NATURAL; break;
        case 2: modificadorEscala = MENOR_ARMONICA;
            for (i=0; i<12; i++){terceras[i] = menorArmonica3[i];}
            for (i=0; i<12; i++){quintas[i]  = menorArmonica5[i];}
            for (i=0; i<12; i++){septimas[i] = menorArmonica7[i];}
            break;
        case 3:
             modificadorEscala = MENOR_MELODICA;
            for (i=0; i<12; i++){terceras[i] = menorMelodica3[i];}
            for (i=0; i<12; i++){quintas[i]  = menorMelodica5[i];}
            for (i=0; i<12; i++){septimas[i] = menorMelodica7[i];}
            break;
        case 4: modificadorEscala = JONICO; break;
        case 5: modificadorEscala = DORICO; break;
        case 6: modificadorEscala = FRIGIO; break;
        case 7: modificadorEscala =  LIDIO; break;
        case 8: modificadorEscala = MIXOLIDIO; break;
        case 9: modificadorEscala =  EOLICO; break;
        case 10: modificadorEscala = LOCRIO; break;
    }
     if(rtdxInputData[1]!=2 && rtdxInputData[1]!=3){
         for (i=0; i<12; i++){terceras[i] = base3[i];}
         for (i=0; i<12; i++){quintas[i]  = base5[i];}
         for (i=0; i<12; i++){septimas[i] = base7[i];}
     }
}
```

Figura 3.74 Código CCS para determinar el modificador por escala y los intervalos del acorde.

rtdxInputData[2]

En función del valor de *rtdxInputData[2]* se determina el modificador por tonalidad en base a la Tabla 3.9 mediante una estructura *switch-case* tal y como muestra la Figura 3.75.

```
if(rtdxInputData[2]!=tono){ // Si no cambia el tono no se hace el switch
   tono = rtdxInputData[2];
   switch (rtdxInputData[2]){
       case 0: modificadorTono = SIs_DO; break;
       case 1: modificadorTono = DOs_REb; break;
       case 2: modificadorTono = RE; break;
       case 3: modificadorTono = REs_MIb; break;
       case 4: modificadorTono = MI_FAb; break;
       case 5: modificadorTono = MIs_FA; break;
       case 6: modificadorTono = FAs_SOLb; break;
       case 7: modificadorTono = SOL; break;
       case 8: modificadorTono = SOLs_LAb; break;
       case 9: modificadorTono = LA; break;
       case 10: modificadorTono = LAs_SIb; break;
       case 11: modificadorTono = SI_DOb; break;
    }
}
```

Figura 3.75 Código en CCS para determinar el modificador por tonalidad.

rtdxInputData[3]

En función del valor de *rtdxInputData[3]* se permite que suene o no una nota que no existe en la escala por lo que sólo afecta a los acordes tríada y cuatríada. Si *rtdxInputData[3]* es 0 la nota no suena, y si es 1, la nota suena pero sin ser armonizada, esto es como si fuera un bypass.

```
…
…
case 5: // Triada
    if(distTercera==0){
        if(rtdxInputData[3]==0){numNotas = 0;}
        else {numNotas = 1;}
    }else{
        f[0] = frecuenciaFundamental;
        f[1] = frecuenciaFundamental * pow(SEMITONO, distTercera);
        f[2] = frecuenciaFundamental * pow(SEMITONO, distQuinta);
        numNotas = 3;
    }
    break;
…
…
```

Figura 3.76 Código en CCS para determinar si la nota fuera de escala suena o no.

rtdxInputData[4]

En función del valor de *rtdxInputData[4]* se determina la ganancia que tendrán las muestras de la señal de entrada antes de calcular la FFT. Como las muestras se deben dividir por el tamaño de la ventana, la ganancia se consigue dividiendo por otro número inferior al tamaño de ésta.

El mínimo valor de ganancia de entrada será igual al tamaño de la ventana, es decir 1024, y la máxima de ganancia será si no se dividen los datos, que es lo mismo que dividir entre 1. Como el valor de *rtdxInputData[4]* está comprendido entre 0 y 100 se debe hacer una conversión de forma que 1 ≤ *Ganancia* ≤ 1024. Para ello se utiliza la ecuación 3.31.

$$Ganancia = a \cdot rtdxInputData[4] + b \tag{3.31}$$

Donde $Ganancia$ es una variable que divide las muestras de entrada y a y b se determinan con el sistema de ecuaciones 3.32 teniendo en cuenta que con el *slider* en reposo *rtdxInputData[4]*=0 y la ganancia de entrada debe ser mínima por lo que el valor debe ser igual a 1024 y a negativa.

$$\begin{cases} 1024 = -a \cdot 0 + b \\ 1 = -a \cdot 100 + b \end{cases} \tag{3.32}$$

Entonces, si se resuelve el sistema a = -10,23 y b = 1024.

rtdxInputData[5]

En función del valor de *rtdxInputData[5]* se reduce o amplifica el valor de la amplitud de la señal de salida para evitar que ésta no se sature o simplemente usarlo como volumen.

Para evitar la saturación de decima el valor de la amplitud hasta 10 veces y para amplificarla se aumenta hasta el doble. Por lo tanto el mínimo valor del volumen multiplica la amplitud por 0,1, y mayor valor de volumen multiplica por 2. Como el valor de *rtdxInputData[5]* está comprendido entre 0 y 100 se debe hacer una conversión de forma que 0,1 ≤ *Volumen* ≤ 2. Para ello se utiliza la ecuación 3.33.

$$Volumen = a \cdot rtdxInputData[5] + b \tag{3.33}$$

Donde $Volumen$ es una variable que multiplica la amplitud de salida y a y b se determinan con el sistema de ecuaciones 3.34 teniendo en cuenta que con el *slider* en reposo *rtdxInputData[5]*=0 cuando el volumen es mínimo y *rtdxInputData[5]*=100 cuando el volumen es máximo.

$$\begin{cases} 0{,}1 = a \cdot 0 + b \\ 2 = a \cdot 100 + b \end{cases} \tag{3.34}$$

Entonces, si se resuelve el sistema a = 0,019 y b = 0,1.

rtdxInputData[6], rtdxInputData[7], rtdxInputData[8] y rtdxInputData[9]

En función del valor de *rtdxInputData[6]* se determina el número de notas de salida que tiene el diseño de armonía. La frecuencia de cada una de ellas se calcula en función del valor de *rtdxInputData[7]*, *rtdxInputData[8]* y *rtdxInputData[9]* en base a la Tabla 3.5.

```
…
…
case 7: // Diseño de armonía
    f[0] = frecuenciaFundamental;
    f[1] = frecuenciaFundamental * pow(SEMITONO, rtdxInputData[7]);
    f[2] = frecuenciaFundamental * pow(SEMITONO, rtdxInputData[8]);
    f[3] = frecuenciaFundamental * pow(SEMITONO, rtdxInputData[9]);
    numNotas = rtdxInputData[6] + 1;
    break;
…
…
```

Figura 3.77 Código en CCS para calcular las frecuencias de las notas según el diseño de armonía.

Capítulo 4. Plan de trabajo y presupuesto

4.1 Plan de trabajo

Debido a la posibilidad que ofrecía este proyecto para hacerlo más o menos completo y a que la herramienta de trabajo es un dispositivo complejo, la dedicación a este proyecto ha sido de dos cuatrimestres. Por lo tanto, el planteamiento de este trabajo se ha realizado teniendo en cuenta esta duración.

Para desarrollar esta aplicación, el trabajo se ha dividido básicamente en dos etapas. El planteamiento de la primera etapa ha sido el siguiente:

- Formación sobre la placa de desarrollo DSK6713 y el software Code Composer Studio.
- Formación sobre teoría y acústica musical.
- Simulación de la aplicación con MATLAB.

Y el planteamiento de la segunda etapa:

- Implementación de la aplicación con el DSP DSK6713.
- Formación sobre teoría y acústica musical (continuación).
- Diseño de una interfaz gráfica con VC++.

Durante la primera etapa se ha asistido como oyente a la clase de la asignatura de *Procesadores digitales* de la titulación de *Sistemas audiovisuales* debido a que en la asignatura de *Dispositivos programables* de la titulación de *Electrónica industrial y automática* sólo se estudian procesadores de propósito general. También se ha debido realizar una formación musical más amplia para comprender mejor la problemática y para desarrollar y generar los efectos de audio. Paralelamente se ha realizado todo el trabajo de simulación con MATLAB.

En la segunda etapa se ha desarrollado el trabajo con el DSP. Por eso, esta parte se ha realizado en el laboratorio de proyectos del departamento de electrónica donde se han dispuesto las herramientas necesarias para el desarrollo de la aplicación como

son la placa de desarrollo DSK7613, el software Code Composer Studio y otros dispositivos como el osciloscopio y el generador de funciones.

Este proyecto se ha realizado entre el 18 de febrero y el 10 diciembre de 2013 con lo que ha tenido una duración aproximada de dos cuatrimestres académicos y un total aproximado de 42 semanas. Si no se tiene en cuenta los períodos vacacionales de semana santa y verano sale un total de 37 semanas.

En la Figura 4.1 se muestra el cronograma por semanas donde se informan sobre cuando se trabajó sobre esa tarea, dando a entender el tiempo aproximado que se ha dedicado a cada parte del proyecto durante el período citado anteriormente.

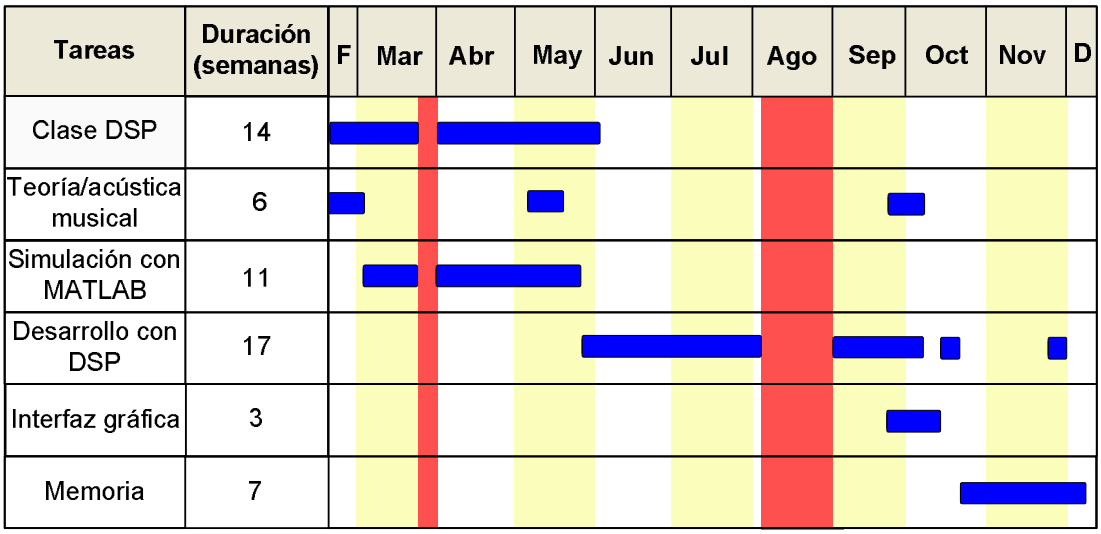


Figura 4.1 Cronograma de las tareas durante la realización del proyecto.

4.2 Presupuesto

La duración del proyecto ha sido de 37 semanas en total. Si para cada semana se tienen en cuenta 40 horas de trabajo sale un total de 1480 horas. Pero para determinar un presupuesto se han considerado únicamente las horas de programación y no las de aprendizaje ni redacción.

El tiempo dedicado a la programación ha sido diferente en las dos etapas del proyecto ya que en la primera ha habido más tiempo de aprendizaje y formación. Teniendo en cuenta esto, para determinar las horas de programación, se considera un tiempo aproximado de 20 horas semanales en la primera etapa y de 30 horas semanales en la segunda.

A continuación se detallan los trabajos y el tiempo que se ha empleado para la programación en el desarrollo de la aplicación en cada etapa.

1º Etapa (MATLAB)	**Horas de programación**
Detección y análisis de la frecuencia fundamental	120
Señal de salida	60
Efectos	40
Subtotal 1	**220**

Tabla 4.1 Horas de programación en la primera etapa.

2ª Etapa (CCS y VC++)	**Horas de programación**
Adaptación de código entre MATLAB y CCS	30
Adquisición de datos	120
Detección y análisis de la frecuencia fundamental	180
Señal de salida	90
Efectos	90
Interfaz gráfica	90
Subtotal 2	**600**

Tabla 4.2 Horas de programación en la segunda etapa.

Por lo tanto, sumando los subtotales de la Tabla 4.1 y la Tabla 4.2 sale un total de 820 horas de programación.

Suponiendo un ingeniero *freelance* (ingeniero que realiza proyectos de forma independiente) se ha calculado el presupuesto en función de lo que se desearía ganar en un mes. Suponiendo 3000 €/brutos mensuales, dividiendo entre 20 días y 8 horas, sale a 22,5 €/h.

Por lo tanto, el presupuesto final es de **18450 €.**

Capítulo 5. Conclusiones

Se ha desarrollado una aplicación con un DSP que permite detectar la frecuencia de una nota y reproducir una armonía en función del efecto musical deseado trabajando en tiempo real. En el caso de los acordes, además se ha tenido en cuenta la escala, la tonalidad y si la nota pertenece o no a la escala. Por lo tanto, el objetivo principal se ha cumplido.

Trabajar en simulación ha permitido desarrollar y probar los algoritmos para determinar la frecuencia y generar los acordes. Además mediante la *toolbox* de procesado de señal de MATLAB se ha podido diseñar un filtro FIR y generar los coeficientes del filtro y del tipo de ventana.

La transformada rápida de Fourier permite conocer el espectro frecuencial de una señal y calcular su frecuencia fundamental. Como no se puede calcular el espectro de un instante determinado, se ha utilizado una ventana que recorre la señal a lo largo del tiempo calculando la FFT de un intervalo de muestras de la señal en cada desplazamiento.

Determinar el tamaño de la ventana es importante, ya que influye en:

- La resolución frecuencial.
- La resolución temporal.
- Tiempo computacional.
- La relación señal/ruido.

La resolución frecuencial y la temporal son inversamente proporcionales por lo que se debe buscar un equilibrio entre ellas ya que con una ventana grande se pueden perder eventualidades temporales y con una ventana pequeña se puede detectar la frecuencia incorrectamente. El equilibrio se ha encontrado reduciendo la frecuencia de muestreo teniendo en cuenta la frecuencia máxima con la que se va a trabajar. Esto ha supuesto utilizar la frecuencia de muestreo más baja del procesador y tener que decimarla para no tener que usar una ventana excesivamente grande.

En la adquisición de datos se ha utilizado la técnica del doble buffer de manera que mientras se están adquiriendo las muestras por un buffer se procesan las del otro garantizando así la ejecución en tiempo real de la aplicación. El tiempo que se están adquiriendo muestras es igual al paso de ventana y es el tiempo que tiene la CPU para procesar los datos. El paso que se ha utilizado es de 8 ms pero se podría aumentar si

el tiempo de procesado fuera insuficiente en el caso de hacer ampliaciones en la aplicación pero nunca deberá ser mayor al tamaño de ventana (256 ms) ya que podría perderse información.

El efecto ventana produce un efecto indeseado que provoca distorsión en el espectro. El uso de una ventana en forma de campana reduce este efecto y mejora el resultado de la FFT. Aun así, queda una pequeña oscilación, pero debido a que hay una resolución de 3,9 Hz, la frecuencia se mantiene estable y no es necesario filtrar ni establecerla a un valor fijo.

Para evitar que el cálculo de la FFT se desborde se deben dividir los datos de entrada por la longitud de la ventana, pero si la señal es pequeña el resultado de la FFT no es bueno ya que los datos están próximos a cero. La solución en estos casos es dividir los datos por una variable que permita modificar su valor según la señal de entrada y permitir que se calcule la FFT correctamente.

Hay armónicos que tienen más amplitud que el fundamental que pueden inducir a error a la hora de determinar la frecuencia de la nota. La solución pasa por potenciar la amplitud del armónico a la frecuencia fundamental. Para ello se ha desarrollado un algoritmo basándose en el algoritmo HPS. Esto se consigue aumentando la amplitud del armónico fundamental con la amplitud del segundo y tercer armónico. De esta manera se ha comprobado que se detecta siempre el armónico fundamental. De todas formas trabajar dentro del espectro no es fácil teniendo en cuenta que al haber una resolución determinada aparecen diferentes barras en el espectro para cada armónico e implica hacer varias operaciones para gestionarlas.

El armonizador puede reproducir hasta siete efectos de armonía distintos en base a la frecuencia fundamental detectada. Todos ellos reproducen la armonía correctamente y en el caso de los acordes se tiene en cuenta la escala y la tonalidad hecho que le otorga calidad al armonizador.

Estudiando la relación que hay entre las escalas y las tonalidades, se han determinado los modificadores oportunos para poder utilizar cualquier escala y tonalidad utilizando sólo tres tablas en vez de 135. Después, se ha desarrollado un algoritmo que permite identificar la nota en función de la frecuencia detectada y determinar el acorde correspondiente en función de la escala y la tonalidad.

Para generar la señal de salida se utiliza una tabla con los datos correspondientes a un periodo de una señal de 1 Hz. A partir de esta tabla se generan el resto de señales. Para evitar que se sature la señal de salida, debido a la suma de señales correspondientes a las notas que forman el armónico, se utiliza una variable a modo de volumen.

El tono de la nota cambia según el tamaño de la ventana, es decir, cada 256 ms, ya que al hacer la FFT de ese período de tiempo sólo se obtiene una frecuencia. Pero en

cambio, la amplitud puede ir cambiando dentro de ese período, por lo que se consigue una buena caracterización de la envolvente ya que la FFT se calcula cada 8 ms.

Los resultados obtenidos han sido satisfactorios ya que la frecuencia detectada ha estado siempre dentro del margen que da la resolución a frecuencias bajas habiendo una variación inferior a 1 Hz y en el caso crítico de 0,4 Hz. Ha habido casos que la variación ha llegado como mucho a 5 Hz a frecuencias más altas, superando el margen que da la resolución, pero este margen sólo es necesario respetarlo a frecuencias bajas, ya que en las frecuencias altas el margen entre notas aumenta y permite una variación cada vez mayor. En el caso de 5 Hz el margen era de 10,3 Hz y en otro caso donde la variación era de 4,8 Hz el margen era de 39,2 Hz.

Debido a que es necesario configurar varios parámetros se ha diseñado una interfaz gráfica que interactúa con la aplicación en tiempo real que permite al usuario más dinamismo a la hora de probar y experimentar con los efectos.

5.1 Limitaciones

Como resultado del desarrollo de esta aplicación se han generado una serie de limitaciones que se deben cumplir para su correcto funcionamiento. Estas son:

- Uso de la placa de desarrollo DSK6713 y el software Code Composer Studio de Texas Instruments para su funcionamiento.
- Necesidad de una interfaz gráfica para gestionar los efectos en tiempo real.
- Se deben tocar las notas de una en una para que no haya un solapamiento de frecuencias.
- El Rango de notas (frecuencias) es de Do_2 (65,4 Hz) a $Fa\#_6$ (1479,9 Hz).
- La resolución frecuencial es de 3,9 Hz por lo que se garantiza un buen resultado a partir de la nota Do_2.
- La resolución temporal es de 256 ms por lo que se pueden tocar 4 notas por segundo.
- Cada nota de salida tiene un único armónico correspondiente al fundamental por lo que el sonido que se reproduce corresponde a una armonía formada por tonos puros.

5.2 Trabajo futuro

El resultado del trabajo realizado cumple con los objetivos propuestos inicialmente. A continuación se proponen una serie de mejoras y ampliaciones del proyecto. Finalmente, en el último punto, se hace referencia a un problema relacionado con el cambio de algoritmo de detección.

Mejorar las prestaciones de la aplicación

Aumentar el rango de frecuencias para poder utilizar más variedad de instrumentos y que se puedan tocar más de cuatro notas por segundo. Esto implica mejorar en resolución frecuencial y modificar la frecuencia de muestreo.

Desarrollar el bloque que detectar la nota en la interfaz gráfica

Este bloque pretende mostrar la frecuencia, la nota y la octava, tanto de la nota que se detecta como las que forman la armonía en tiempo real. Pero a pesar que la comunicación está preparada y se ha comprobado que se puede usar en los dos sentidos, se han detectado problemas a la hora de de enviar los datos desde el DSP al PC.

Interfaz Hardware

La interfaz gráfica asistida por PC implica dejar de tocar en algún momento para cambiar la configuración. Se propone como alternativa a la interfaz gráfica desarrollar una interfaz hardware mediante pedales o botones de rápido acceso que permitan al músico cambiar o configurar efectos tocando en directo. Además, con un pedal se podría añadir otro efecto que permita modificar la frecuencia de la nota según se pisa el pedal.

Caracterizar el timbre del instrumento

Se podrían añadir armónicos para caracterizar el instrumento.

Por un lado se podría reproducir el timbre del instrumento ya que durante la aplicación sólo se han generado con tonos puros.

Por otro lado, en el caso de tenerlo caracterizado y sintetizado, se podría usar independientemente del instrumento de entrada, de forma que se podría estar tocando un instrumento y sonando otro, es decir, se podría tocar una guitarra y sonar una trompeta. De esta manera se le ofrecería al músico la posibilidad de “tocar” un instrumento que no domina.

Uso del algoritmo de detección basado en el algoritmo HPS

Antes de modificar el algoritmo, a pesar de que no se detectaba bien la frecuencia fundamental, el sonido de la señal de salida era bueno.

El hecho de cambiar el código para solucionar el problema de la frecuencia fundamental ha provocado un efecto indeseable. Este efecto hace que la señal de salida, aunque respeta el tono y la envolvente, en ocasiones pase a valer cero o prácticamente cero durante un tiempo muy pequeño pero perceptible al oído.

En función de la frecuencia de la señal de entrada, este problema se da con más o menos frecuencia y hay veces que no ocurre.

La causa del problema no se ha podido detectar pero si se ha comprobado que el tiempo que la señal de salida no es correcta es múltiplo del paso de ventana. Esto hizo pensar que como el nuevo algoritmo realizaba más operaciones que el algoritmo original, la CPU no tenía tiempo de realizar todo el procesado ya que el tiempo máximo que tiene para hacerlo es igual al paso. Pero a pesar de incrementar el paso para dar más tiempo a la CPU el problema persistía.

El programa antes de cambiar el algoritmo funcionaba correctamente con un paso de 8 ms. Incluso funcionaba bien con un paso de 2 ms para todos los efectos excepto para los acordes ya que implicaban más calculo. En cambio con el nuevo algoritmo se han probado pasos de hasta 256 ms y el problema continuaba igual. No se ha podido dar más tiempo ya que 256 ms es igual al tamaño de la ventana y hacer un paso más grande implicaría perder eventualidades temporales. En todos los casos que se han probado la duración de este efecto indeseable coincidía con el paso. Después de observar que no se mejoraba la salida aumentando el paso, éste se dejó a 8 ms.

A parte de incrementar el paso también se han sacado graficas del vector y los buffers de salida reiteradas veces y en distintos instantes de tiempo y en el resultado no se ha apreciado este problema.

Por lo tanto, este tema queda para un futuro análisis con más detenimiento, pero en cualquier caso no es un problema excesivamente grave ya que no siempre ocurre y los efectos se reproducen correctamente por lo que mientras tanto es preferible mantener el nuevo algoritmo que el anterior.

Capítulo 6. Bilbiografía

[1] Basso, G. Análisis espectral: *La transformada de Fourier en la música*. 2ª ed. La Plata: Ediciones Al Margen, 2001. ISBN 950-34-0150-X.

[2] Barrero, F.; Toral, S.; Ruiz, M. *Procesadores Digitales de Señal de altas prestaciones de Texas Instruments: De la familia TMS320C3x a la TMS320C6000*. Madrid: McGraw-Hill, 2005. ISBN 84-481-9834-4.

[3] Bernal, J.; Gómez, P.; Bobadilla, J. Una visión práctica en el uso de la transformada de Fourier como herramienta para el análisis espectral de la voz. Universidad Politécnica de Madrid, Departamento de Informática Aplicada y Departamento de Arquitectura y Tecnología de Sistemas Informáticos.

[4] Chassaing, R. *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. New Jersey: John Wiley & Sons, Inc., 2005. ISBN 0-471-69007-4.

[5] Cordantonopulos, V. Curso completo de Teoría de la Música. En: *pianoaventura* [en línea]. 2002 [Consulta: Febrero 2013]. Disponible en: <http://www.pianoaventura.com/aprendizaje-del-piano.php>

[6] Everest, F.; Polhmann, K. *The Master Handbook Of Acoustics*. 5th ed. McGraw-Hill. 2009. ISBN 978-0-07-160333-1.

[7] Florez, E.; Cardona, S.; Jordi, L. Selección de la ventana temporal en la transformada de Fourier en tiempos cortos utilizada en el análisis de señales de vibración para determinar planos en las ruedas de un tren. *Rev. Fac. Ing. Univ. Antioquía*. 2009, no. 50, p 145-158.

[8] Frecuencia de las notas musicales. En: *La tecla de escape* [en línea]. 2013 [Consulta: Febrero 2013]. Disponible en: <http://latecladeescape.com/t/Frecuencia+de+las+notas+musicales>

[9] Gala, O.; Morillo, J.; García, A. Optimización de aplicaciones en arquitecturas DSP. Proyecto final de carrera, Universidad Complutense de Madrid, Departamento de Arquitectura de Computadores y Automática, 2006.

[10] Gómez, E. Representación de señales de audio. Síntesis y procesamiento del sonido I, Escola Superior de Música de Catalunya, Departamento de Sonología, 2009.

[11] Grijota, J. Implementación de un procesador digital de efectos mediante DSP e interfaz gráfica sobre plataforma W2000 y XP. Proyecto final de carrera, UPC, Departamento de Electrónica, 2008.

[12] Haykin, S.; Van Veen B. *Señales y sistemas*.1ª ed. Ciudad de México: Editorial Limusa, 2001. ISBN 968-18-5914-6.

[13] Károlyi, O. *Introducción a la música.* Madrid: Alianza Editorial, 2003. ISBN: 978-84-206-3526-2.

[14] Luzuriaga, J.; Pérez, R. O. *La física de los instrumentos musicales*. 1ª ed. Buenos Aires: Eudeba, 2007. ISBN 978-950-23-1459-4.

[15] Masip, A. Ruido en la medida del sensor y filtrado. 2013.

[16] Mateu, M. A. *Armonía práctica. Volumen 1.* Valencia: Ab Música Ediciones Musicales, 2006. ISBN 84-609-2448-3.

[17] Moron, J. *Señales y sistemas*. Vereda del Lago: Fondo Editorial Biblioteca Universidad Rafael Urdaneta, 2011. ISBN 978-980-7131-06-3.

[18] Panello, D. Apunte de Visual C++ [en línea]. [Consulta: Septiembre 2013]. Disponible en: <http://www.dcp.com.ar/mfc/pagina1.htm#WorkSpace>

[19] Pérez, A. Mejora de un conversor de audio MIDI e implementación en tiempo real. Proyecto final de carrera, UPC, Departamento de Teoría de la Señal y Comunicaciones, 2007.

[20] Raja, M. Teoría musical. En: *Aprende gratis* [en línea]. 2007 [Consulta: Febrero 2013]. Disponible en: <http://www.aprende-gratis.com/teoria-musical/>

[21] Rogalla, H. RTDX Tutorial Version 1.0 [en línea]. 2005 [Consulta: Septiembre 2013]. Disponible en: <http://www.tsseshop.com/Developer/Tutorials/RTDX/TutorialRTDX.html>

[22] Soyuz. Tabla: rango de frecuencias de los instrumentos musicales. En: *hispasonic* [en línea]. 2002 [Consulta: Febrero 2013]. Disponible en: <http://www.hispasonic.com/reportajes/tabla-rango-frecuencias-instrumentos-musicales/39>

[23] Texas Instruments. TLV320AIC23B Stereo Audio CODEC, 8- to 96-hHz, With Integrated Headphone Amplifier. Data Manual. No: SLWS106H, February 2004.

[24] Texas Instruments. TMS320 DSP/BIOS v5.42. User's Guide. No: SPRU423I, August 2012.

[25] Texas Instruments. TMS320C6000 Chip Support Library API. Reference Guide. No: SPRU401J, August 2004.

[26] Texas Instruments. TMS320C6000 DSK Board Support Library API. User's Guide. No: SPRU432A, October 2001.

[27] Texas Instruments. TMS320C6000 DSP Enhanced Direct Memory Acces (EDMA) Controller. Reference Guide. No: SPRU234C, November 2006.

[28] Texas Instruments. TMS320C6000 DSP Multichannel Buffered Serial Port (McBSP). Reference Guide. No: SPRU580G, December 2006.

[29] Texas Instruments. TMS320C6000 DSP/BIOS 5.x Application Programming Interface (API). Reference Guide. No: SPRU403S, August 2012.

[30] Texas Instruments. TMS320C62x DSP Library. Programmer's Reference. No: SPRU402B, October 2003.

[31] Texas Instruments. TMS320C6713 DSK. Technical Reference. Rev. B, November 2003.

[32] The MathWorks. Digital Signal Processing Toolbox For Use with MATLAB. User's Guide. Version 5. 2000.

[33] Vazquez, E. Desarrollo de aplicaciones en el DSK TMS320C6711. Tesis doctoral, Universisdad Autónoma del estado de Hidalgo, 2007.

[34] Wikipedia. Pitch shift. En: *Wikipedia. The Free Encyclopedia* [en línea]. 2013 [Consulta: Diciembre 2013]. Disponible en: <http://en.wikipedia.org/wiki/Pitch_shift>

[35] Wikipedia. Eventide, Inc. En: *Wikipedia. The Free Encyclopedia* [en línea]. 2013 [Consulta: Diciembre 2013]. Disponible en: <http://en.wikipedia.org/wiki/Eventide%2c_Inc#H910_Harmonizer>

[36] Zamacois, J. *Teoría de la música (I).* Madrid: Mundimúsica Ediciones, 2007. ISBN 978-84-8236-253-4.

[37] Zamacois, J. *Tratado de armonía. Libro I.* Cooper City: SpanPress Universitaria, 1997. ISBN 1-58045-911-0.

Apéndices

Apéndice A. Tablas de frecuencias de las notas musicales

Do_0: 16,351598 Hz	Do_1: 32,703196 Hz	Do_2: 65,406391 Hz	Do_3: 130,812783 Hz
$Do\#_0$: 17,323914 Hz	$Do\#_1$: 34,647829 Hz	$Do\#_2$: 69,295658 Hz	$Do\#_3$: 138,591315 Hz
Re_0: 18,354048 Hz	Re_1: 36,708096 Hz	Re_2: 73,416192 Hz	Re_3: 146,832384 Hz
$Re\#_0$: 19,445436 Hz	$Re\#_1$: 38,890873 Hz	$Re\#_2$: 77,781746 Hz	$Re\#_3$: 155,563492 Hz
Mi_0: 20,601722 Hz	Mi_1: 41,203445 Hz	Mi_2: 82,406889 Hz	Mi_3: 164,813778 Hz
Fa_0: 21,826764 Hz	Fa_1: 43,653529 Hz	Fa_2: 87,307058 Hz	Fa_3: 174,614116 Hz
$Fa\#_0$: 23,124651 Hz	$Fa\#_1$: 46,249303 Hz	$Fa\#_2$: 92,498606 Hz	$Fa\#_3$: 184,997211 Hz
Sol_0: 24,499715 Hz	Sol_1: 48,999429 Hz	Sol_2: 97,998859 Hz	Sol_3: 195,997718 Hz
$Sol\#_0$: 25,956544 Hz	$Sol\#_1$: 51,913087 Hz	$Sol\#_2$: 103,826174 Hz	$Sol\#_3$: 207,652349 Hz
La_0: 27,500000 Hz	La_1: 55,000000 Hz	La_2: 110,000000 Hz	La_3: 220,000000 Hz
$La\#_0$: 29,135235 Hz	$La\#_1$: 58,270470 Hz	$La\#_2$: 116,540940 Hz	$La\#_3$: 233,081881 Hz
Si_0: 30,867706 Hz	Si_1: 61,735413 Hz	Si_2: 123,470825 Hz	Si_3: 246,941651 Hz

Do_4: 261,625565 Hz	Do_5: 523,251131 Hz	Do_6: 1046,502261 Hz	Do_7: 2093,004522 Hz
$Do\#_4$: 277,182631 Hz	$Do\#_5$: 554,365262 Hz	$Do\#_6$: 1108,730524 Hz	$Do\#_7$: 2217,461048 Hz
Re_4: 293,664768 Hz	Re_5: 587,329536 Hz	Re_6: 1174,659072 Hz	Re_7: 2349,318143 Hz
$Re\#_4$: 311,126984 Hz	$Re\#_5$: 622,253967 Hz	$Re\#_6$: 1244,507935 Hz	$Re\#_7$: 2489,015870 Hz
Mi_4: 329,627557 Hz	Mi_5: 659,255114 Hz	Mi_6: 1318,510228 Hz	Mi_7: 2637,020455 Hz
Fa_4: 349,228231 Hz	Fa_5: 698,456463 Hz	Fa_6: 1396,912926 Hz	Fa_7: 2793,825851 Hz
$Fa\#_4$: 369,994423 Hz	$Fa\#_5$: 739,988845 Hz	$Fa\#_6$: 1479,977691 Hz	$Fa\#_7$: 2959,955382 Hz
Sol_4: 391,995436 Hz	Sol_5: 783,990872 Hz	Sol_6: 1567,981744 Hz	Sol_7: 3135,963488 Hz
$Sol\#_4$: 415,304698 Hz	$Sol\#_5$: 830,609395 Hz	$Sol\#_6$: 1661,218790 Hz	$Sol\#_7$: 3322,437581 Hz
La_4: 440,000000 Hz	La_5: 880,000000 Hz	La_6: 1760,000000 Hz	La_7: 3520,000000 Hz
$La\#_4$: 466,163762 Hz	$La\#_5$: 932,327523 Hz	$La\#_6$: 1864,655046 Hz	$La\#_7$: 3729,310092 Hz
Si_4: 493,883301 Hz	Si_5: 987,766603 Hz	Si_6: 1975,533205 Hz	Si_7: 3951,066410 Hz

Apéndice B. Código MATLAB (simulación)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                                 %
%                 SIMULACION DE ARMONIZADOR CON MATLAB                            %
%                                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% AUTOR: Carlos Anchuela Arnalte.
% PROYECTO: Armonizador para instrumento musical.
% NOMBRE DEL ARCHIVO: armSIM.m

clc;        % Se borra el historial de comandos.
close all;  % Se cierran todas las ventanas.
clear all;  % Se borran todas las variables.


%% INICIALIZACION DE VARIABLES

tiempo = 0; % Tiempo transcurrido.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  PASO  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
paso = 33;        % Paso de ventana.
contMuestras = 0; % Contador de muestras transcurridas hasta el Paso.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  VENTANA  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
longVentana = 1024;            % Longitud de Ventana.
ventana=zeros(longVentana,1); % Vector con las ultimas n muestras de
                               % la señal de entrada a analizar.
tipoVentana = @hamming;        % Tipo de Ventana
coefVentana = window(tipoVentana ,longVentana); % Coef. de la ventana.

%%%%%%%%  DETECTAR FRECUENCIA FUNDAMENTAL Y AMPLITUD MAXIMA  %%%%%%%%%
frecFund = 0;    % Frecuencia fundamental.
amplitudMax = 0; % Amplitud a la frecuencia fundamental.

%%%%%%%%%%%%%%%%%%%%%%%%  FILTRO DE PRIMER ORDEN  %%%%%%%%%%%%%%%%%%%%%%%%
frecFundFil1 = 0; % Frec. fundamental filtrada con filtro de 1er orden
Lambda = 0.9;     % Grado de filtraje del filtro.
resultado1=0;     % Resultado después de filtrar.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  FILTRO FIR  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
frecFundFil2 = 0; % Frecuencia fundamental filtrada con filtro FIR.
ordenFiltro = 32; % Orden del filtro.
Wn = 0.1;         % Frecuencia de corte del filtro donde 0<Wn<1.
                  % Para Wn=1 la frecuencia de corte = fs/2.
coefFiltro = fir1(ordenFiltro, Wn);    % Coeficientes del filtro.
muestrasFrecFund=zeros(ordenFiltro,1); % Vector con las ultimas n
                                       % muestras de la frecuencia
                                       % fundamental.

%%%%%%%%%%%%%%%%%%%%%%%%  ESTABLECER FRECUENCIA  %%%%%%%%%%%%%%%%%%%%%%%%
semitono = 2^(1/12); % Distancia mínima entre dos semitonos.
contFFT = 0;         % Contador de veces que se realiza la FFT.
frecNota = 0;        % Frecuencia establecida.
frecEst = 0;         % Indicador de frec. establecida (0=NO, 1=SI).
```

```
numFFT = 32;          % Numero de FFT consecutivas para establecer la
                      % frecuencia de una nota.

%%%%%%%%%%%%%%%%%%%%%%%%%  FRECUENCIAS ACORDE  %%%%%%%%%%%%%%%%%%%%%%%%%
f1 = 0;                % Frecuencia de la nota establecida.
f3 = 0;                % Frecuencia de la tercera.
f5 = 0;                % Frecuencia de la quinta.
f7 = 0;                % Frecuencia de la séptima
fLow = 0;              % Frec. a la octava más baja para la guitarra.
fComp = 80.0943175;    % Frecuencia mínima de comparación del grupo de
                       % notas (Do, Re, Mi...) en sus octavas más bajas
                       % para la guitarra.

%%%%%%%%%%%%%%%%%%%%%%%%%  ESCALA DO MAYOR  %%%%%%%%%%%%%%%%%%%%%%%%%
% Distancias en semitonos, respecto la nota detectada, para formar acordes:
%             80 Hz ...              ... 110 Hz ...                 ...
160 Hz
% POSICION=  1    2    3    4    5    6    7    8    9   10   11   12
%    NOTA = Mi   Fa   Fa#  Sol  Sol# La   La#  Si   Do  Do#  Re   Re#
tercerasDo=[ 3    4    0    4    0    3    0    3    4    0    3   0];
% TERCERA = Sol  La        Si        Do        Re   Mi        Fa
quintasDo =[ 7    7    0    7    0    7    0    6    7    0    7   0];
%  QUINTA = Si   Do        Re        Mi        Fa   Sol       La
septimasDo=[10   11    0   10    0   10    0   10   11    0   10   0];
% SEPTIMA = Re   Mi        Fa        Sol       La   Si        Do

tercerasEscala = tercerasDo;
quintasEscala = quintasDo;
septimasEscala = septimasDo;


%% LECTURA DEL FICHERO WAV Y REPRODUCCION DE LA SEÑAL DE ENTRADA

% Adquisición de la señal de entrada:
[entrada fs nBits] = wavread('Mi3-Do3-Fa3-La3.wav');

% Se reproduce la señal de entrada:
sound(entrada, fs);


%% REPRODUCCION DE LA SEÑAL DE SALIDA

for N=1:length(entrada)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    LECTURA DE LA SEÑAL DE ENTRADA                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Las muestras de entrada se van guardando una a una en un vector
    % de longitud igual a la ventana a medida que pasa el tiempo:
    for j=1:longVentana-1
        ventana(j,1) = ventana(j+1);
    end
    % Se lee una nueva muestra de entrada:
    ventana(longVentana) = entrada(N);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                   PASO                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Se incrementa el contador de muestras transcurridas entrada mira
    % si son igual al paso (o la longitud de la ventana para la
    % primera vez).Si es así, se calcula la frecuencia fundamental de
    % la nota. Si no se ha llegado al paso no se calcula nada.
    contMuestras = contMuestras+1;
    if contMuestras>=paso && N>=longVentana

        % Se inicializa el contador para realizar la siguiente FFT:
        contMuestras = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          DETECTAR FRECUENCIA FUNDAMENTAL Y AMPLITUD MAXIMA                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % Si la amplitud de la señal es cero (con un margen de +/-
        % 0.05) en TODAS las muestras del vector de entrada no se
        % realiza la FFT. (Hacer esto supone mucho tiempo
        % computacional).
        if ventana<=0.05 & ventana>=-0.05
            frecFund = 0;
            frecNota = 0;
             f1 = 0;
             f3 = 0;
             f5 = 0;
             f7 = 0;
        % En caso contrario:
        else
            % Se cambia la ventana por una en forma de campana:
            for j=1:longVentana
                ventanaTipo(j) = coefVentana(j)*ventana(j);
            end
            % Se calcula la FFT de la ventana:
            FFT = abs(fft(ventanaTipo, fs));
            % Se calcula la frecuencia fundamental. Cuando se detecta
            % la máxima amplitud espectral del resultado de la FFT, el
            % índice "j" corresponde a la frecuencia fundamental:
            amplitudMax = 0;
            for j=70:1500 %%% length(FFT)/2
                if FFT(j)>amplitudMax
                    amplitudMax = FFT(j);
                    frecFund = j;
                end
            end
        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   FILTRO DE PRIMER ORDEN O DE MEDIA MOVIL CON PESOS EXPONENCIALES  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%         resultado1 = (Lambda*resultado1) + ((1-Lambda)*frecFund);
%%%         frecFundFil1 = resultado1;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          FILTRO FIR PASABAJOS DE LA FRECUENCIA FUNDAMETNAL          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        for j=ordenFiltro:-1:2
            muestrasFrecFund(j) = muestrasFrecFund(j-1);
        end
        muestrasFrecFund(1) = frecFund;

        resultado2=0;
        for j=1:ordenFiltro
            resultado2 = resultado2 +
                        coefFiltro(j)*muestrasFrecFund(j);
        end
        frecFundFil2 = resultado2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    ESTABLECER FRECUENCIA NOTA                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        frec = frecFund;     % Frecuencia fundamental sin filtrar.
        frec = frecFundFil1; % Frec. fundamental filtro 1er orden.
        frec = frecFundFil2;     % Frecuencia fundamental filtro FIR.

        %Se establece la frecuencia inicial de comparación:
        if contFFT==0
            frecFundInicial = frec;
        end
        % Se mira si la frecuencia actual respecto a la inicial esta
        % dentro del margen valido de frecuencias:
        if frecFundInicial<frec*semitono && frecFundInicial>frec/semitono
            if frecEst==0
                % Si esta dentro, se incrementa el contador y se mira
                % si han pasado el numero de FFT para establecer la
                % frecuencia de la nota:
                contFFT = contFFT+1;
                if contFFT==numFFT
                    frecNota = frec;
                    frecEst = 1;
                end
            end
        % Si no está dentro se vuelve a poner el contador de FFT a
        % cero y la nota queda como no establecida:
        else
            contFFT = 0;
            frecEst = 0;
        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          FRECUENCIAS ACORDE                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % Si la frecuencia de la nota es menor que la mínima para la
        % guitarra, o no ha cambiado, no se calcula nada:
        if frecNota>fComp && frecNota~=fLow

            fLow = frecNota; % Frecuencia de la nota establecida.
```

```
            % Se cambia el valor de la frecuencia de la nota al de su
            % octava más baja posible para la guitarra:
            while fLow>fComp
                fLow = fLow/2;
            end
            fLow = fLow*2;

            posicion = 0; % Índice de los vectores para las distancias
                          % de terceras, quintas y séptimas.

            % Se busca la ubicación de la nota dentro del grupo de
            % notas de frecuencia más baja para la guitarra:
            while fLow>fComp
                posicion = posicion+1;
                fLow = fLow/semitono;
            end

            % Distancia en semitonos respecto la nota establecida:
            distTercera = tercerasEscala(posicion); % para la tercera
            distQuinta = quintasEscala(posicion);   % para la quinta
            distSeptima = septimasEscala(posicion); % para la séptima

            % Se calcula la frecuencia de cada nota del acorde:
            f1 = frecNota;                       % Frec. de la nota
            f3 = frecNota*semitono^distTercera;  % Frec. de su 3a.
            f5 = frecNota*semitono^distQuinta;   % Frec. de su 5a.
            f7 = frecNota*semitono^distSeptima;  % Frec. de su 7a.

        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%             FIN DE LOS CALCULOS CUANDO SE LLEGA AL PASO                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              SEÑAL DE SALIDA                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Se calcula el tiempo según la muestra actual:
    tiempo=N/fs;

    % Se calcula la salida de cada nota del acorde según la muestra
    % actual. La amplitud se divide entre 100 porque para reproducir
    % la señal debe  estar entre -1 y 1 ya que sino la señal se
    % satura:
       NOTA = (amplitudMax/100)*sin(2*pi*f1*tiempo);
    TERCERA = (amplitudMax/100)*sin(2*pi*f3*tiempo);
     QUINTA = (amplitudMax/100)*sin(2*pi*f5*tiempo);
    SEPTIMA = (amplitudMax/100)*sin(2*pi*f7*tiempo);

    % Se calcula la salida según el efecto:
    salidaBypass(N) = NOTA;
    salidaTriada(N) = NOTA + TERCERA + QUINTA;
    salidaCuatriada(N) = NOTA + TERCERA + QUINTA + SEPTIMA;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            VECTORES PARA PLOTEAR                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    t(N) = N;               % Tiempo transcurrido.

    fn(N) = frecNota;       % Frecuencia establecida.
    ff(N) = frecFund;       % Frecuencia fundamental.
    ff1(N) = frecFundFil1;  % Frec. fundamental con filtro 1er orden.
    ff2(N) = frecFundFil2;  % Frecuencia fundamental con filtro FIR.

    frec0(N) = f1;          % Frecuencia de la nota establecida.
    frec3(N) = f3;          % Frecuencia de su tercera.
    frec5(N) = f5;          % Frecuencia de su quinta.
    frec7(N) = f7;          % Frecuencia de su séptima.

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                         REPRODUCIR Y GUARDAR SALIDA                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Se reproduce la señal de salida.
sound(salidaBypass, fs);
sound(salidaTriada, fs);
sound(salidaCuatriada, fs);

% Se guarda la señal de salida en un fichero WAV.
wavwrite(salidaBypass,fs,'salidaBypass.wav');
wavwrite(salidaTriada,fs,'salidaTriada.wav');
wavwrite(salidaCuatriada,fs,'salidaCuatriada.wav');
```

```
%% PLOTS

% SEÑAL DE ENTRADA Y DE SALIDA
figure;
subplot(211);
plot(entrada);
title('SEÑAL DE ENTRADA');
xlabel('Tiempo (muestras)');
ylabel('Amplitud');
subplot(212);
plot(salidaBypass);
title('SEÑAL DE SALIDA');
xlabel('Tiempo (muestras)');
ylabel('Amplitud');

% FRECUENCIA FUNDAMENTAL
figure;
plot(t,fn,'blue',t,ff,'green',t,ff2,'red');
title('FRECUENCIA FUNDAMENTAL');
xlabel('Tiempo (muestras)');
ylabel('Frecuencia (Hz)');

% FRECUENCIAS ACORDE
figure;
plot(t,frec0,'blue',t,frec3,'green',t,frec5,'red',t,frec7,'black')
title('FRECUENCIAS ACORDE');
```

```
xlabel('Tiempo (muestras)');
ylabel('Frecuencia (Hz)');

% TIPO DE VENTANA
figure;
plot(coefVentana);
title('TIPO DE VENTANA');
xlabel('Posición');
ylabel('Factor');
```

Apéndice C. Código MATLAB (coeficientes FIR)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                                 %
%          GENERADOR DE COEFICIENTES PARA EL TIPO DE VENTANA                      %
%                                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% AUTOR: Carlos Anchuela Arnalte.
% PROYECTO: Armonizador para instrumento musical.
% NOMBRE DEL ARCHIVO: coefVentana.m

clc;
close all;
clear all;

longVentana = 1024;     % Longitud de Ventana
tipoVentana = @hamming; % Tipo de Ventana

% Otros tipos de ventanas:
% @bartlett       - Bartlett window.
% @barthannwin    - Modified Bartlett-Hanning window.
% @blackman       - Blackman window.
% @blackmanharris - Minimum 4-term Blackman-Harris window.
% @bohmanwin      - Bohman window.
% @chebwin        - Chebyshev window.
% @flattopwin     - Flat Top window.
% @gausswin       - Gaussian window.
% @hamming        - Hamming window.
% @hann           - Hann window.
% @kaiser         - Kaiser window.
% @nuttallwin     - Nuttall defined minimum 4-term Blackman-Harris
%                   window.
% @parzenwin      - Parzen (de la Valle-Poussin) window.
% @rectwin        - Rectangular window.
% @taylorwin      - Taylor window.
% @tukeywin       - Tukey window.
% @triang         - Triangular window.

% Se llena el vector con los coeficientes de la ventana:
coef = window(tipoVentana ,longVentana);

% Se pasan los datos a short:
for j=1:longVentana
    coefVentana(j) = round(coef(j)*32767);
end

% Se genera un fichero .h para usar los coeficientes con código C:
dlmwrite('hamming.h',coefVentana,',');
```

Apéndice D. Código Code Composer Studio

Este apéndice también se incluye el código de Texas Instrument que se ha utilizado para desarrollar la aplicación.

Armonizador.c

```
/*********************************************************************/
/*********************************************************************/
/***                                                               ***/
/***              ARMONIZADOR PARA INSTRUMENTO MUSICAL             ***/
/***                                                               ***/
/*********************************************************************/
/*********************************************************************/

// AUTOR: Carlos Anchuela Arnalte.
// PROYECTO: Armonizador para instrumento musical.
// NOMBRE DEL ARCHIVO: Armonizador.c


/*====================================================================*/
/*====================================================================*/
/*==                                                                ==*/
/*==                      FICHEROS DE CABECERA                      ==*/
/*==                                                                ==*/
/*====================================================================*/
/*====================================================================*/

#include <math.h>            //Librería para operaciones matemáticas
#include <csl.h>             //Librería CSL
#include <csl_irq.h>         //Interrupciones
#include <csl_mcbsp.h>       //Comunicación serie
#include <csl_edma.h>        //Acceso directo a memoria
#include <dsk6713.h>         //Librería BSL
#include <dsk6713_aic23.h>   //Códec de audio
#include <rtdx.h>            //Librería para comunicación RTDX
#include <tw_radix2.h>       //Coeficientes w para calcular FFT radix2
#include <bitrev_index.h>    //Coeficientes para ordenar el resultado
                             //de la FFT
#include <dsp_radix2.h>      //Calcular la FFT
#include <dsp_bitrev_cplx.h> //Ordenar la FFT
#include <armonizador.h>     //Prototipos de funciones
#include <FIR_41.h>          //Coeficientes del filtro FIR de orden 41
#include <hamming.h>         //Coeficientes de la ventana hamming
#include <acordes.h>         //Tablas para generar acordes


/*====================================================================*/
/*====================================================================*/
/*==                                                                ==*/
/*==                   DECLARACION DE CONSTANTES                    ==*/
/*==                                                                ==*/
/*====================================================================*/
/*====================================================================*/

#define ORDEN 41             //Orden del filtro FIR
#define DECIMACION 2         //Decimación de la frecuencia de muestreo
#define FS_IN 8000           //Frecuencia de muestreo de entrada
#define FS_OUT 8000          //Frecuencia de muestreo de salida
```

```
#define VENTANA 1024            //Tamaño del vector de muestras de
entrada
#define PASO 64              //Numero de muestras de solo un canal
#define SCALE 32767.5        //Factor de escala para pasar a short
#define LONG_RTDX 20         //Tamaño de la array de comunicación RTDX
#define LONG_INDEX 32   //longIndex=(int)ceil(sqrt((double)VENTANA))
#define LONG_TABLA 8000 //Muestras para reconstruir un periodo
#define McBSP1_DRR 0x01900000 //Dirección del registro de recepción
#define McBSP1_DXR 0x01900004 //Dirección del registro de salida
#define NOTAS 12              //Numero de notas musicales
#define NOTAS_ACORDE 4  //Numero máximo de notas que se pueden generar
#define FREC_MIN 63.57090194  //Frec mínima de comparación de notas
#define FREC_MAX 1500         //Frecuencia máxima de la aplicación
#define SEMITONO 1.059463094  //Es igual a la raíz doceava de 2
#define PI 3.14159265358979323846 //Número PI
#define NUM_IND 10      //Numero de barras del espectro a analizar
#define RUIDO 60


/*====================================================================*/
/*====================================================================*/
/*==                                                                ==*/
/*==                  DECLARACION DE VARIABLES                      ==*/
/*                                                                  ==*/
/*====================================================================*/
/*====================================================================*/

int i, j; //Variables para bucles for

/*--------------------------------------------------------------------*/
/*                 VARIABLES PARA COMUNICACION RTDX                   */
/*--------------------------------------------------------------------*/

float rtdxOutputData[LONG_RTDX];
int rtdxInputData[LONG_RTDX];
//rtdxInputData[0] - efecto
//rtdxInputData[1] - modificador por escala
//rtdxInputData[2] - modificador por nota
//rtdxInputData[3] - ON/OFF de nota fuera de escala para acordes
//rtdxInputData[4] - ganancia de entrada
//rtdxInputData[5] - volumen/ctrl de saturación
//rtdxInputData[6] - numero de intervalos para crear armonía
//rtdxInputData[7] - distancia en semitonos del primer intervalo
//rtdxInputData[8] - distancia en semitonos del segundo intervalo
//rtdxInputData[9] - distancia en semitonos del tercer intervalo

//Canal para recibir del VC++:
RTDX_CreateInputChannel (rtdxInputChannel);
//Canal para enviar al VC++:
RTDX_CreateOutputChannel (rtdxOutputChannel);

/*--------------------------------------------------------------------*/
/*                VARIABLES PARA FILTRAR Y DECIMAR                    */
/*--------------------------------------------------------------------*/

short numFIR;              //Muestras de entrada que se van a filtrar
short indiceMuestrasEntradaFIR; //Índice de la muestra que se va a
filtrar
short  muestrasEntradaFIR[ORDEN+PASO];//Vector  con  las  ultimas  n
muestras
int   muestraSalidaFIR;
```

```
/*-------------------------------------------------------------------*/
/*  VARIABLES PARA CALCULAR LA FREC. FUNDAMENTAL Y AMPLITUD MAXIMA  */
/*-------------------------------------------------------------------*/

float coefGainIn;//coeficiente de conversión para rango de 1 a VENTANA
float gainIn;    //Variable que divide las muestras para la FFT
float coefVolumen; //coeficiente de conversión para rango de 0.1 a 2
float volumen;     //variable que multiplica la amplitud de salida
short coefRadix2[VENTANA];     //Coeficientes w para radix 2
short index[VENTANA];          //Coeficientes para ordenar la FFT
short muestrasEntrada[VENTANA];//Muestras filtradas y decimadas
                               //de entrada de uno de los canales
short muestrasFFT[2*VENTANA];  //Muestras para hacer la FFT
                               //(incluye parte real e imaginaria)

int amplitudEspectralMax;
short indiceAmplitudEspectralMax;
float resolucionFrecuencia;
float frecuenciaFundamental;

short indices[NUM_IND];  //Índices de las amplitudes máximas de la FFT
int   amplitudes[NUM_IND];     //Amplitudes máximas de la FFT
int   moduloFFT[VENTANA/2];    //Modulo al cuadrado de la FFT
int   auxiliar[(3*VENTANA)/2]; //espectro auxiliar
short iMin;                    //Índice de a la frec. más baja
short iMax;                    //Índice de a la frec. mas alta

/*-------------------------------------------------------------------*/
/*                VARIABLES PARA GENERAR EFECTOS                     */
/*-------------------------------------------------------------------*/

short efecto;          //Mira si ha cambiado el efecto
short escala;          //mira si ha cambiado la escala
short tono;            //Mira si ha cambiado la tonalidad
short modificadorEscala;
short modificadorTono;
short octava;
float frecNota;
short indiceNota;      //Índice correspondiente a la nota
short indiceNota2;     //Índice modificado según escala y tonalidad
short distTercera;     //Distancia en semitonos para la tercera
short distQuinta;      //Distancia en semitonos para la quinta
short distSeptima;     //Distancia en semitonos para la séptima
short terceras[NOTAS]; //Intervalos de tercera
short quintas[NOTAS];  //Intervalos de quinta
short septimas[NOTAS]; //Intervalos de séptima
float f[NOTAS_ACORDE]; //Frecuencia de las notas de la armonía

/*-------------------------------------------------------------------*/
/*           VARIABLES PARA GENERAR LA SEÑAL DE SALIDA               */
/*-------------------------------------------------------------------*/

short tabla[LONG_TABLA];            //Tabla para reconstruir señal
float frecNaturalTabla;             //Frecuencia de la tabla
float incrementoTabla[NOTAS_ACORDE]; //Incremento en la tabla
float indiceTabla[NOTAS_ACORDE];     //Índice de la tabla calculado
```

```
short indT[NOTAS_ACORDE];           //Valor entero del índice
short numNotas;                     //Numero de notas de la armonía
int   salida[PASO];                 //Señal de salida


/*---------------------------------------------------------------------*/
/*    VARIABLES PARA LOS BUFFERS DE ENTRADA Y SALIDA DE DATOS          */
/*---------------------------------------------------------------------*/

short  cicloBuffers;       //Indica en que ciclo nos encontramos
short  bufferIn1[2*PASO]; //Buffer 1 de recepción de datos
short  bufferIn2[2*PASO]; //Buffer 2 de recepción de datos
short  bufferOut1[2*PASO]; //Buffer 1 de salida de datos
short  bufferOut2[2*PASO]; //Buffer 2 de salida de datos

/*---------------------------------------------------------------------*/
/*                    VARIABLES DE INTERRUPCION                        */
/*---------------------------------------------------------------------*/

short flagEDMAinterrupt; //Indica si se ha producido una interrupción

/*---------------------------------------------------------------------- */
/*                  VARIABLES PARA EL CODEC DE AUDIO                   */
/*---------------------------------------------------------------------*/

DSK6713_AIC23_CodecHandle hCodec; //Handle para el códec AIC23

DSK6713_AIC23_Config codecConfig = {
      0x0017, //0 LEFTINVOL Left line input channel volume
      0x0017, //1 RIGHTINVOL Right line input channel volume
      0x01F9, //2 LEFTHPVOL Left channel headphone vol
      0x01F9, //3 RIGHTHPVOL Right channel headphone vol
      0x0011, //4 ANAPATH Analog audio path control
      0x0000, //5 DIGPATH Digital audio path control
      0x0000, //6 POWERDOWN Power down control
      0x0043, //7 DIGIF Digital audio interface format
      0x000D, //8 SAMPLERATE Sample rate control. IN:8KHz/OUT:8KHz
      0x0001  //9 DIGACT Digital interface activation
   };

/---------------------------------------------------------------------*/
/*            VARIABLES PARA LOS CANALES DEL EDMA                      */
/*---------------------------------------------------------------------*/

//Handles para los canales de lectura y escritura del EDMA:
EDMA_Handle hEdmaLeer, hEdmaEscribir;
//Handles para los canales que desplazan y actualizan el vector
//muestrasFIR:
EDMA_Handle hEdmaDesplazarEntradaFIR, hEdmaActualizarEntradaFIR;
//Handle para el canal que desplaza el vector muestrasEntrada:
EDMA_Handle hEdmaDesplazarEntrada;
//Handle para el canal que copia el vector muestrasEntrada en el
//vector muestrasFFT:
EDMA_Handle hEdmaCopiarEntrada;
//Handles para los canales de recarga del EDMA para leer:
EDMA_Handle hEdmaLINKleer1, hEdmaLINKleer2;
//Handles para los canales de recarga del EDMA para escribir:
EDMA_Handle hEdmaLINKescribir1, hEdmaLINKescribir2;
//Handles para los canales de recarga del EDMA para actualizar la
//muestras a filtrar:
EDMA_Handle hEdmaLINKactualizarEntradaFIR1,
```

```
            hEdmaLINKactualizarEntradaFIR2;



/*======================================================================*/
/*======================================================================*/
/*==                                                                  ==*/
/*==           CONFIGURACION DE LOS CANALES DEL EDMA                  ==*/
/*==                                                                  ==*/
/*======================================================================*/
/*======================================================================*/

/*----------------------------------------------------------------------*/
/*   CONFIG. DE LOS CANALES PARA LOS BUFFERS DE ENTRADA Y SALIDA        */
/*----------------------------------------------------------------------*/

EDMA_Config edma_Config_Leer1 = {
  EDMA_OPT_RMK(               //OPTIONS:
    EDMA_OPT_PRI_HIGH,        // Prioridad alta
    EDMA_OPT_ESIZE_16BIT,     // Longitud de los datos a leer
    EDMA_OPT_2DS_NO,          // Origen de una dimensión
    EDMA_OPT_SUM_NONE,        // Dirección fija (DRR MCBSP1)
    EDMA_OPT_2DD_NO,          // Destino de una dimensión
    EDMA_OPT_DUM_INC,         // Se incrementa una posicion en el
                              // buffer al leer
    EDMA_OPT_TCINT_YES,       // Se habilita el chain con el canal 8
    EDMA_OPT_TCC_OF(8),       // Cuando se llena el buffer se pasa al
                              // canal 8
    EDMA_OPT_LINK_YES,        // Se permite el linkado para los canales
                              // de recarga
    EDMA_OPT_FS_NO            // Canal esta sincronizado por elemento
                              // (dato que se lee del McBSP)
   ),
  EDMA_SRC_OF(McBSP1_DRR),    //SOURCE: DRR MCBSP1
  EDMA_CNT_OF(2*PASO),        //LENGHT: Longitud de datos
  EDMA_DST_OF(bufferIn1),     //DESTINATION: bufferIn1[0]
  EDMA_IDX_OF(0),             //INDEX: No se usa. Se deja a 0
                              //(dirección del siguiente elemento sin
                              //offset --> posiciones contiguas)
  EDMA_RLD_OF(0)              //RELOAD: link. Se pone 0 porque mas
                              //adelante se configurara la recarga
 };

EDMA_Config edma_Config_Leer2 = {
  EDMA_OPT_RMK(               //OPTIONS: Se configura igual que
                              //edma_Config_Leer1
    EDMA_OPT_PRI_HIGH,
    EDMA_OPT_ESIZE_16BIT,
    EDMA_OPT_2DS_NO,
    EDMA_OPT_SUM_NONE,
    EDMA_OPT_2DD_NO,
    EDMA_OPT_DUM_INC,
    EDMA_OPT_TCINT_YES,
    EDMA_OPT_TCC_OF(8),
    EDMA_OPT_LINK_YES,        // Se permite el linkado. Se linka el
                              // canal de recarga leer2 con el canal de
                              // recarga leer1
    EDMA_OPT_FS_NO
   ),
  EDMA_SRC_OF(McBSP1_DRR),    //SOURCE: DRR MCBSP1
```

```
  EDMA_CNT_OF(2*PASO),        //LENGHT: Longitud de datos
  EDMA_DST_OF(bufferIn2),     //DESTINATION: bufferIn2[0]
  EDMA_IDX_OF(0),             //INDEX: No se usa. Se deja a 0
                              //(dirección del siguiente elemento sin
                              //offset --> posiciones contiguas)
  EDMA_RLD_OF(0)              //RELOAD: link. Se pone 0 porque mas
                              //adelante se configurara la recarga
 };

EDMA_Config edma_Config_Escribir1 = {
  EDMA_OPT_RMK(               //OPTIONS:
    EDMA_OPT_PRI_HIGH,        // Prioridad alta
    EDMA_OPT_ESIZE_16BIT,     // Longitud de los datos a escribir
    EDMA_OPT_2DS_NO,          // Origen de una dimensión (elemento
                              // dentro de un frame)
    EDMA_OPT_SUM_INC,         // Se incrementa una posicion de memoria
                              // en el buffer al escribir un dato
    EDMA_OPT_2DD_NO,          // Destino de una dimensión (elemento
                              // dentro de un frame)
    EDMA_OPT_DUM_NONE,        // Dirección fija porque siempre se
                              // escribe del mismo sitio (DXR MCBSP1)
    EDMA_OPT_TCINT_NO,        // No se habilita interrupción en modo
                              // escritura
    EDMA_OPT_TCC_OF(6),       // Como no se interrumpe ni se hace chain
                              // se puede usar cualquier canal. En este
                              // caso se usa el 6
    EDMA_OPT_LINK_YES,        // Se permite el linkado. Se linka el
                              // canal de recarga escribir1 con el
                              // canal de recarga escribir2
    EDMA_OPT_FS_NO            // Como 2DS y 2DD es NO y FS es NO, el
                              // canal esta sincronizado por elemento
                              // (dato que se escribe en el McBSP)
   ),
  EDMA_SRC_OF(bufferOut1),    //SOURCE: bufferOut1[0]
  EDMA_CNT_OF(2*PASO),        //LENGHT: Longitud de 64 datos
  EDMA_DST_OF(McBSP1_DXR),    //DESTINATION: DXR MCBSP1
  EDMA_IDX_OF(0),             //INDEX: No se usa. Se deja a 0
                              //(dirección del siguiente elemento sin
                              //offset --> posiciones contiguas)
  EDMA_RLD_OF(0)              //RELOAD: link. Se pone 0 porque mas
                              //adelante se configurara la recarga
 };

EDMA_Config edma_Config_Escribir2 = {
  EDMA_OPT_RMK(               //OPTIONS: Se configura igual que
                              //edma_Config_Escribir1
    EDMA_OPT_PRI_HIGH,
    EDMA_OPT_ESIZE_16BIT,
    EDMA_OPT_2DS_NO,
    EDMA_OPT_SUM_INC,
    EDMA_OPT_2DD_NO,
    EDMA_OPT_DUM_NONE,
    EDMA_OPT_TCINT_NO,
    EDMA_OPT_TCC_OF(6),
    EDMA_OPT_LINK_YES,        // Se permite el linkado. Se linka el
                              // canal de recarga escribir2 con el
                              // canal de recarga escribir1
    EDMA_OPT_FS_NO
   ),
  EDMA_SRC_OF(bufferOut2),    //SOURCE: bufferOut2[0]
  EDMA_CNT_OF(2*PASO),        //LENGHT: Longitud de datos
```

```
  EDMA_DST_OF(McBSP1_DXR),    //DESTINATION: DXR MCBSP1
  EDMA_IDX_OF(0),             //INDEX: No se usa. Se deja a 0
                              //(dirección del siguiente elemento sin
                              //offset --> posiciones contiguas)
  EDMA_RLD_OF(0)              //RELOAD: link. Se pone 0 porque mas
                              //adelante se configurara la recarga
 };

/*----------------------------------------------------------------------*/
/*         CONFIGURACION DE LOS CANALES PARA EL FILTRO FIR              */
/*----------------------------------------------------------------------*/

EDMA_Config edma_Config_DesplazarEntradaFIR = { //CANAL 8
  EDMA_OPT_RMK(               //OPTIONS:
    EDMA_OPT_PRI_LOW,         // Prioridad baja
    EDMA_OPT_ESIZE_16BIT,     // Longitud de los datos que se desplazan
    EDMA_OPT_2DS_YES,         // Origen de dos dimensiones (frame
                              // dentro de un bloque)
    EDMA_OPT_SUM_INC,         // Se incrementa una posicion de memoria
                              // en el vector al leer el dato
    EDMA_OPT_2DD_YES,         // Destino de dos dimensiones (frame
                              // dentro de un bloque)
    EDMA_OPT_DUM_INC,         // Se incrementa una posicion de memoria
                              // en el vector al escribir el dato
    EDMA_OPT_TCINT_YES,       // Se habilita el chain para poder
                              // encadenar con el siguiente canal (hacer
                              // chain) que en este caso es el canal 9
    EDMA_OPT_TCC_OF(9),       // Cuando se han desplazado los datos del
                              // vector se pasa al canal 9 (para el
                              //  chain solo se pueden usar los canales
                              // 8,9,10 y 11)
    EDMA_OPT_LINK_NO,         // No hay linkado
    EDMA_OPT_FS_NO            // Como 2DS y 2DD es YES y FS es NO
                              // el canal esta sincronizado por array.
                              // Se mueve el array entero sin
                              // interrupciones
   ),
  EDMA_SRC_OF(muestrasEntradaFIR+PASO),//SOURCE: muentrasFIR[PASO]
  EDMA_CNT_OF(ORDEN),              //LENGHT: Longitud de datos
                                   //igual al orden del filtro:
                                   //(ORDEN+PASO)-PASO = ORDEN
  EDMA_DST_OF(muestrasEntradaFIR), //DESTINATION: muentrasFIR[0]
  EDMA_IDX_OF(0),       //INDEX: No se usa. Se deja a 0 (dirección del
                        //siguiente elemento sin offset --> posiciones
                        //contiguas)
  EDMA_RLD_OF(0)        //RELOAD: link. Se pone 0 porque más adelante
                        //se configurara la recarga
 };

EDMA_Config edma_Config_ActualizarEntradaFIR1 = { //CANAL 9
  EDMA_OPT_RMK(               //OPTIONS:
    EDMA_OPT_PRI_LOW,         // Prioridad baja
    EDMA_OPT_ESIZE_16BIT,     // Longitud de los datos que se
                              // actualizan
    EDMA_OPT_2DS_NO,          // Origen de una dimensión (elemento
                              // dentro de un frame)
    EDMA_OPT_SUM_IDX,   // Se incrementa tantas posiciones de memoria
                        //(1 posición son 4 bytes) en el vector al leer
                        // el dato según se indica en INDEX
    EDMA_OPT_2DD_NO,    // Destino de una dimensión (elemento dentro de
                        // un frame)
```

```
    EDMA_OPT_DUM_INC,   // Se incrementa una posicion de memoria en el
                        // vector al escribir el dato
    EDMA_OPT_TCINT_YES, // Se habilita interrupción para que la CPU
                        // procese datos y haga cálculos
    EDMA_OPT_TCC_OF(12),// Como se interrumpe pero no se hace
                        // chain se puede usar cualquier canal menos el
                        // 8,9,10 o 11. En este caso se usa el 12
    EDMA_OPT_LINK_YES,  // Se permite el linkado. Se linka el canal de
                        // recarga actualizar1 con el canal de recarga
                        // actualizar2
    EDMA_OPT_FS_YES     // Como 2DS y 2DD es NO y FS es YES el canal
                        // esta sincronizado por frame. Se mueve el
                        // array entero sin interrupciones
   ),
  EDMA_SRC_OF(bufferIn1+1),  //SOURCE: bufferIn1[0]
  EDMA_CNT_OF(PASO),    // LENGHT: Longitud de datos igual al PASO.
                        // Sólo se usa el canal izquierdo o el derecho
  EDMA_DST_OF(muestrasEntradaFIR+ORDEN),//DESTINATION: muentrasFIR[ORDEN]
  EDMA_IDX_ELEIDX_OF(4),//INDEX: Se lee/escribe el dato (elemento)
                        //cada 2 posiciones (4 bytes). Sólo se usa el
                        //canal izquierdo o el derecho
  EDMA_RLD_OF(0)        //RELOAD: link. Se pone 0 porque más adelante
                        //se configurara la recarga
 };

 EDMA_Config edma_Config_ActualizarEntradaFIR2 = { //CANAL 9
   EDMA_OPT_RMK(        //OPTIONS: Se configura igual que
                        //edma_Config_ActualizarEntradaFIR1
     EDMA_OPT_PRI_LOW,
     EDMA_OPT_ESIZE_16BIT,
     EDMA_OPT_2DS_NO,
     EDMA_OPT_SUM_IDX,
     EDMA_OPT_2DD_NO,
     EDMA_OPT_DUM_INC,
     EDMA_OPT_TCINT_YES,
     EDMA_OPT_TCC_OF(12),
     EDMA_OPT_LINK_YES, // Se permite el linkado. Se linka el canal de
                        // recarga actualizar2 con el canal de recarga
                        // actualizar1
     EDMA_OPT_FS_YES
    ),
   EDMA_SRC_OF(bufferIn2+1), //SOURCE: bufferIn2[0]
   EDMA_CNT_OF(PASO),   //LENGHT: Longitud de datos igual al PASO.
                        //Sólo se usa el canal izquierdo o el derecho
   EDMA_DST_OF(muestrasEntradaFIR+ORDEN),//DESTINATION: muentrasFIR[ORDEN]
   EDMA_IDX_ELEIDX_OF(4), //INDEX: Se lee/escribe el dato (elemento)
                        //cada 2 posiciones (4 bytes). Sólo se usa el
                        //canal izquierdo o el derecho
   EDMA_RLD_OF(0)       //RELOAD: link. Se pone 0 porque más adelante
                        //se configurara la recarga
 };

/*----------------------------------------------------------------------*/
/*             CONFIGURACION DE LOS CANALES PARA LA FFT                 */
/*----------------------------------------------------------------------*/

EDMA_Config edma_Config_DesplazarEntrada = { //CANAL 4
  EDMA_OPT_RMK(              //OPTIONS:
    EDMA_OPT_PRI_LOW,        // Prioridad baja
    EDMA_OPT_ESIZE_16BIT,    // Longitud de los datos que se desplazan
    EDMA_OPT_2DS_YES,        // Origen de dos dimensiones (frame
```

```
                              // dentro de un bloque)
    EDMA_OPT_SUM_INC,         // Se incrementa una posicion de memoria
                              // en el vector al leer el dato
    EDMA_OPT_2DD_YES,         // Destino de dos dimensiones (frame
                              // dentro de un bloque)
    EDMA_OPT_DUM_INC,   // Se incrementa una posicion de memoria
                        // en el vector al escribir el dato
    EDMA_OPT_TCINT_YES, // Se habilita el chain para poder encadenar
                        // con el siguiente canal (hacer chain) que en
                        // este caso es el canal 9
    EDMA_OPT_TCC_OF(10),// Cuando se han desplazado los datos del
                        // vector se pasa al canal 9 (para el chain
                        // solo se pueden usar los canales 8,9,10 y 11)
    EDMA_OPT_LINK_NO,   // No hay linkado
    EDMA_OPT_FS_NO      // Como 2DS y 2DD es YES y FS es NO el canal
                        // esta sincronizado por array. Se mueve el
                        // array entero sin interrupciones
   ),
  EDMA_SRC_OF(muestrasEntrada+(PASO/DECIMACION)),//SOURCE:
                                  //muentrasEntrada[PASO/DECIMACION]
  EDMA_CNT_OF(VENTANA-(PASO/DECIMACION)),//LENGHT: Longitud de datos
                                  //igual al tamaño de la VENTANA
                                  //menos el PASO/DECIMACION
  EDMA_DST_OF(muestrasEntrada),     //DESTINATION: muentrasEntrada[0]
  EDMA_IDX_OF(0), //INDEX: No se usa. Se deja a 0 (dirección del
                  //siguiente  elemento  sin  offset  -->  posiciones
                  //contiguas)
  EDMA_RLD_OF(0)  //RELOAD: link. Se pone 0 porque más adelante se
                  //configurara la recarga
 };

EDMA_Config edma_Config_CopiarEntrada = { //CANAL 10
  EDMA_OPT_RMK(               //OPTIONS:
    EDMA_OPT_PRI_LOW,         // Prioridad baja
    EDMA_OPT_ESIZE_16BIT,     // Longitud de los datos que se
                              // actualizan
    EDMA_OPT_2DS_NO,          // Origen de una dimensión (elemento
                              // dentro de un frame)
    EDMA_OPT_SUM_INC,         // Se incrementa una posicion de memoria
                              // en el vector al leer el dato
    EDMA_OPT_2DD_NO,          // Destino de una dimensión (elemento
                              // dentro de un frame)
    EDMA_OPT_DUM_IDX,         // Se incrementa tantas posiciones de
                              // memoria (1 posición son 4 bytes) en el
                              // vector al leer el dato según se indica
                              // en INDEX
    EDMA_OPT_TCINT_NO,        // No se habilita interrupción
    EDMA_OPT_TCC_OF(5),       // Como no se interrumpe ni se hace chain
                              // se puede usar cualquier canal. En este
                              // caso se usa el 6
    EDMA_OPT_LINK_NO,         // No hay linkado
    EDMA_OPT_FS_YES           // Como 2DS y 2DD es NO y FS es YES el
                              // canal esta sincronizado por array. Se
                              // mueve el array entero sin
                              // interrupciones
   ),
  EDMA_SRC_OF(muestrasEntrada),//SOURCE: muestrasEntrada[0]
  EDMA_CNT_OF(VENTANA),      //LENGHT: Longitud de datos a mover igual
                              //al tamaño de la VENTANA
  EDMA_DST_OF(muestrasFFT),   //DESTINATION: muentrasFFT[0]
  EDMA_IDX_ELEIDX_OF(4),//INDEX: Se lee/escribe el dato (elemento)
```

```
                        //cada 2 posiciones (4 bytes). Sólo se usa el
                        //canal izquierdo o el derecho
  EDMA_RLD_OF(0)        //RELOAD: link. Se pone 0 porque mas
                        //adelante se configurara la recarga
 };


/*=====================================================================*/
/*=====================================================================*/
/*==                                                                 ==*/
/*==                       PROGRAMA PRINCIPAL                        ==*/
/*==                                                                 ==*/
/*=====================================================================*/
/*=====================================================================*/

void main(){

/*---------------------------------------------------------------------*/
/*                    NICIALIZACION DE VARIABLES                       */
/*---------------------------------------------------------------------*/

   /* Comunicación RTDX */

   RTDX_enableInput(&rtdxInputChannel);  //Habilitar canal de entrada
   RTDX_enableOutput(&rtdxOutputChannel);//Habilitar canal de salida

   for (i=0; i<LONG_RTDX; i++) {rtdxOutputData[i]=0;}
   for      (i=0;      i<LONG_RTDX;      i++)      {rtdxInputData[i]=0;}

   rtdxInputData[5] = 48; //De 0 a 100 (0=0.1, 48=1.012, 100=2)
   rtdxInputData[6] = 1;  //De 1 a 3.
   rtdxInputData[7] = -12;//De -12 a +12
   rtdxInputData[8] = 7;  //De -12 a +12
   rtdxInputData[9] = 12; //De -12 a +12
   rtdxInputData[10] = 0;

   /* Filtrar y decimar */
   numFIR = PASO/DECIMACION;
   indiceMuestrasEntradaFIR = 0;
   for (i=0; i<(ORDEN+PASO); i++) {muestrasEntradaFIR[i]=0;}
   muestraSalidaFIR = 0;

   /* Frecuencia y amplitud */
   coefGainIn = (1-(float)VENTANA)/100; //Para rango de VENTANA a 1
   gainIn = VENTANA;
   coefVolumen = (2-0.1)/100;           //Para rango de 0.1 a 2
   volumen = 1;
   for (i=0; i<VENTANA; i++) {coefRadix2[i]=0;}
   for (i=0; i<LONG_INDEX; i++) {index[i]=0;}
   for (i=0; i<VENTANA; i++) {muestrasEntrada[i]=0;}
   for (i=0; i<(2*VENTANA); i++) {muestrasFFT[i]=0;}

   amplitudEspectralMax = 0;
   indiceAmplitudEspectralMax = 0;
   resolucionFrecuencia =
      ((float)FS_IN/(float)DECIMACION)/((float)VENTANA);
   frecuenciaFundamental = 0;

   for (i=0; i<NUM_IND; i++) {indices[i]=0;}
   for (i=0; i<NUM_IND; i++) {amplitudes[i]=0;}
   for (i=0; i<VENTANA/2; i++) {moduloFFT[i]=0;}
```

```
   for (i=0; i<(3*VENTANA)/2; i++) {auxiliar[i]=0;}
   iMin = (FREC_MIN/resolucionFrecuencia)-1;
   iMax = VENTANA/2;


/* Efectos */
   efecto = -1;
   escala = -1;
   tono = -1;
   modificadorEscala = 0;
   modificadorTono = 0;
   for (i=0; i<NOTAS; i++) {terceras[i] = 0;}
   for (i=0; i<NOTAS; i++) {quintas[i]  = 0;}
   for (i=0; i<NOTAS; i++) {septimas[i] = 0;}
   octava = 0;
   frecNota = 0;
   indiceNota = 0;
   indiceNota2 = 0;
   distTercera = 0;
   distQuinta = 0;
   distSeptima = 0;
   for(i=0; i<NOTAS_ACORDE; i++){f[i] = 0;}

   /* Salida */
   //Tabla de sinus para generar señales de salida. Se multiplica por
   //32767 para pasar a short:
   for (i=0; i<LONG_TABLA; i++){tabla[i] =
      sin((2*PI*i)/LONG_TABLA)*(32767);}
   frecNaturalTabla = (float)FS_OUT/(float)LONG_TABLA;
   for(i=0; i<NOTAS_ACORDE; i++){incrementoTabla[i] = 0;}
   for(i=0; i<NOTAS_ACORDE; i++){indiceTabla[i] = 0;}
   for(i=0; i<NOTAS_ACORDE; i++){indT[i] = 0;}
   numNotas = 1;
   for (i=0; i<(PASO); i++) {salida[i]=0;}

   /* Buffers */
   cicloBuffers = 0;
   for (i=0; i<(2*PASO); i++) {bufferIn1[i]=0;}
   for (i=0; i<(2*PASO); i++) {bufferIn2[i]=0;}
   for (i=0; i<(2*PASO); i++) {bufferOut1[i]=0;}
   for (i=0; i<(2*PASO); i++) {bufferOut2[i]=0;}

   /* Interrupciones */
   flagEDMAinterrupt = 0;

/*--------------------------------------------------------------------*/
/*                    FUNCIONES DE INICIALIZACION                     */
/*--------------------------------------------------------------------*/

   CSL_init();     //Inicializar la CSL
   DSK6713_init(); //Función para inicializar la BSL
   hCodec = DSK6713_AIC23_openCodec(0, &codecConfig);//Función para
                                                     //abrir el Códec
   edma_init();    //Rutina para inicialización del EDMA

/*--------------------------------------------------------------------*/
/*              GENERAR COEFICIENTES PARA CALCULAR LA FFT              */
/*--------------------------------------------------------------------*/

   //Generación del vector coefRadix2 para realizar la FFT radix2:
   gen_twiddle(coefRadix2, VENTANA, SCALE);
```

```
   //Generación del vector index para ordenar la FFT:
   bitrev_index(index, VENTANA);



/*----------------------------------------------------------------------*/
/*    ROPORCIONAR EVENTOS PARA LANZAR LAS TRANSFERENCIAS POR EDMA   */
/*----------------------------------------------------------------------*/

   EDMA_setChannel(hEdmaLeer);
   EDMA_setChannel(hEdmaEscribir);
   EDMA_setChannel(hEdmaDesplazarEntradaFIR);
   EDMA_setChannel(hEdmaActualizarEntradaFIR);
   EDMA_setChannel(hEdmaDesplazarEntrada);
   EDMA_setChannel(hEdmaCopiarEntrada);

/*----------------------------------------------------------------------*/
/*             HABILITACION DE LA INTERRUPCION DEL EDMA             */
/*----------------------------------------------------------------------*/


   IRQ_globalEnable();
   IRQ_enable(IRQ_EVT_EDMAINT);


/*======================================================================*/
/*======================================================================*/
/*==                                                                  ==*/
/*==                       BUCLE INFINITO                             ==*/
/*==                                                                  ==*/
/*======================================================================*/
/*======================================================================*/

   while(1) {

/*----------------------------------------------------------------------*/
/*                          COMUNICACION RTDX                           */
/*----------------------------------------------------------------------*/


        if(!RTDX_channelBusy(&rtdxInputChannel)){ //Mira si el canal
            //esta libre y se queda esperando a que le lleguen datos
            //pero el código se sigue ejecutando
            RTDX_readNB(&rtdxInputChannel, &rtdxInputData,
                        sizeof(rtdxInputData));

            //Si cambia el efecto se inicializan los índices:
                  if(rtdxInputData[0]!=efecto){
               efecto = rtdxInputData[0];
               for(i=0; i<NOTAS_ACORDE; i++){
                   indiceTabla[i] = 0;
                   indT[i] = 0;
               }
            }

            //Se actualizan los coeficientes:
            gainIn = (coefGainIn * (float)rtdxInputData[4]) + VENTANA;
            volumen = (coefVolumen * (float)rtdxInputData[5]) + 0.1;
        }
```

```
/*-------------------------------------------------------------------*/
/*                        INTERRUPCION EDMA                          */
/*-------------------------------------------------------------------*/

/*
   Las acciones que se realizan son:
       - Filtrar y decimar
       - Calcular la FFT
       - Calcular frec. fundamental y amplitud del primer armónico
       - Determinar el acorde
       - Calcular las frecuencias de las notas según el  efecto
       - Determinar la salida
*/

        if (flagEDMAinterrupt==1) {
            flagEDMAinterrupt = 0;

/*-------------------------------------------------------------------*/
/*                         FILTRAR Y DECIMAR                         */
/*-------------------------------------------------------------------*/

            indiceMuestrasEntradaFIR = ORDEN;
            for (i=0; i<numFIR; i++){
                muestraSalidaFIR = 0;
                for (j=0; j<ORDEN; j++) {
                  muestraSalidaFIR = muestraSalidaFIR + (coefFIR[j]*
                  muestrasEntradaFIR[indiceMuestrasEntradaFIR-j]);
                }
                muestraSalidaFIR = muestraSalidaFIR>>15;
                //Prepara el índice de la siguiente muestra a filtrar:
                indiceMuestrasEntradaFIR = indiceMuestrasEntradaFIR +
                                           DECIMACION;
                //Actualiza muestrasEntrada con la muestra filtrada:
                muestrasEntrada[VENTANA-numFIR+i] = muestraSalidaFIR;
                //Actualiza muestrasFFT con la muestra filtrada:
                muestrasFFT[2*(VENTANA-numFIR+i)] = muestraSalidaFIR;
            }

/*-------------------------------------------------------------------*/
/*                          CALCULAR FFT                             */
/*-------------------------------------------------------------------*/


              /* Preparar FFT */
            for (i=0;i<VENTANA;i++) {
                //Parte real:
                //Se divide entre la longitud de la ventana para que
                //no se desborde el resultado al hacer la FFT. Se
                //multiplica por coefVentana para mejorar el
                //resultado de la FFT. Se desplaza 15 posiciones para
                //que quepa en un short
                muestrasFFT[2*i] =
                 ((muestrasFFT[i*2]/(short)gainIn)*coefVentana[i])>>15;
                //Parte imaginaria:
                  muestrasFFT[2*i+1] = 0; //Se pone a cero
            }
```

```
            /* Calcular FFT */
            DSP_radix2 (VENTANA, muestrasFFT, coefRadix2);


            /* Ordenar FFT */
            DSP_bitrev_cplx ((int *)muestrasFFT, index, VENTANA);

            /* Calcular modulo al cuadrado de la FFT */
            for (i=iMin; i<iMax; i++) {
                moduloFFT[i] = (muestrasFFT[2*i]*muestrasFFT[2*i]) +
                               (muestrasFFT[2*i+1]*muestrasFFT[2*i+1]);
                auxiliar[i] = 0;
            }

/*-------------------------------------------------------------------*/
/*      CALCULAR FREC FUND Y AMPLITUD DEL PRIMER ARMONICO            */
/*-------------------------------------------------------------------*/


            // Seleccionar los índices de las amplitudes mas grandes
            for(i=0; i<NUM_IND; i++){
                amplitudes[i] = 0;
                    for (j=iMin; j<iMax; j++) {
                        if (moduloFFT[j]>amplitudes[i]) {
                            amplitudes[i] = moduloFFT[j];
                            indices[i] = j;
                        }
                    }
                moduloFFT[indices[i]] = 0;
            }

            // Igualar amplitudes de posiciones contiguas
            for(i=0; i<NUM_IND; i++){
                //restaurar amplitud original en moduloFFT:
                moduloFFT[indices[i]] = amplitudes[i];
                for(j=0; j<NUM_IND; j++){
                    if(indices[i]+1==indices[j]
                                    || ind ices[i]-1==indices[j]
                                    || indices[i]+2==indices[j]
                                    || indices[i]-2==indices[j]){
                        if(amplitudes[i]<amplitudes[j]){
                            amplitudes[i]=amplitudes[j];
                        }
                     }
                 }

                 //Colocar las amplitudes de los índices elegidos en
                 //el vector auxiliar
                 auxiliar[indices[i]-3] = amplitudes[i];
                 auxiliar[indices[i]-2] = amplitudes[i];
                 auxiliar[indices[i]-1] = amplitudes[i];
                 auxiliar[indices[i]] = amplitudes[i];
                 auxiliar[indices[i]+1] = amplitudes[i];
                 auxiliar[indices[i]+2] = amplitudes[i];
                 auxiliar[indices[i]+3] = amplitudes[i];
            }

            // Amplificar amplitud fundamental con 2do y 3er armónico
            for (i=0; i<NUM_IND; i++){
                auxiliar[indices[i]] = auxiliar[indices[i]] +
```

```
            auxiliar[indices[i]*2] + auxiliar[indices[i]*3];
        }

        // Eliminar valores fuera del rango
        auxiliar[iMin-3]=0;
        auxiliar[iMin-2]=0;
        auxiliar[iMin-1]=0;
        auxiliar[iMax+1]=0;
        auxiliar[iMax+2]=0;
        auxiliar[iMax+3]=0;

        // Buscar la amplitud máxima
        amplitudEspectralMax = 0;
        indiceAmplitudEspectralMax = 0;
        for (i=0; i<(NUM_IND); i++) {
            if (auxiliar[indices[i]]>amplitudEspectralMax) {
               amplitudEspectralMax = auxiliar[indices[i]];
               indiceAmplitudEspectralMax = indices[i];
            }
        }

        // Calcular amplitud y frecuencia fundamental
        amplitudEspectralMax =
              sqrt((double)moduloFFT[indiceAmplitudEspectralMax]);
        frecuenciaFundamental =
         (float)indiceAmplitudEspectralMax * resolucionFrecuencia;

        if (amplitudEspectralMax<RUIDO){
              amplitudEspectralMax=0;
              frecuenciaFundamental=0;
        }else{
              amplitudEspectralMax = volumen *
                                     amplitudEspectralMax;
        }
        if (frecuenciaFundamental<FREC_MIN){
              frecuenciaFundamental=0;
              amplitudEspectralMax = 0;
        }
        if (frecuenciaFundamental>FREC_MAX){
              frecuenciaFundamental=0;
              amplitudEspectralMax = 0;
         }

/*---------------------------------------------------------------------*/
/*     HABILITAR EVENTO PARA ENCADENAR CON EL CANAL 4 DEL EDMA      */
/*---------------------------------------------------------------------*/

        EDMA_setChannel(hEdmaDesplazarEntrada);

/*---------------------------------------------------------------------*/
/*                          ACORDE INTELIGENTE                         */
/*---------------------------------------------------------------------*/

/*   Calcular distancia notas para modo acorde tríada y cuatríada   */

        if (rtdxInputData[0]==5 || rtdxInputData[0]==6){

           //Se determina el modificador por escala y se llenan
           //los vectores terceras, quintas y séptimas. Si no
           //cambia la escala no se hace el switch
           if(rtdxInputData[1]!=escala){
```

```
    escala = rtdxInputData[1];
    switch (rtdxInputData[1]){
       case 0: modificadorEscala = MAYOR; break;
       case 1: modificadorEscala = MENOR_NATURAL;
               break;
       case 2:
           modificadorEscala = MENOR_ARMONICA;
           for (i=0; i<12; i++){
               terceras[i] = menorArmonica3[i];}
           for (i=0; i<12; i++){
               quintas[i]  = menorArmonica5[i];}
           for (i=0; i<12; i++){
               septimas[i] = menorArmonica7[i];}
           break;
       case 3:
           modificadorEscala = MENOR_MELODICA;
           for (i=0; i<12; i++){
               terceras[i] = menorMelodica3[i];}
           for (i=0; i<12; i++){
               quintas[i]  = menorMelodica5[i];}
           for (i=0; i<12; i++){
               septimas[i] = menorMelodica7[i];}
           break;
       case 4: modificadorEscala = JONICO; break;
       case 5: modificadorEscala = DORICO; break;
       case 6: modificadorEscala = FRIGIO; break;
       case 7: modificadorEscala =  LIDIO; break;
       case 8: modificadorEscala = MIXOLIDIO; break;
       case 9: modificadorEscala =  EOLICO; break;
       case  10:  modificadorEscala  =  LOCRIO;  break;

    }
    if(rtdxInputData[1]!=2 && rtdxInputData[1]!=3){
       for (i=0; i<12; i++){terceras[i] = base3[i];}
       for (i=0; i<12; i++){quintas[i]  = base5[i];}
       for (i=0; i<12; i++){septimas[i] = base7[i];}
    }
}

//Se determina el modificador por tonalidad
//Si no cambia el tono no se hace el switch
if(rtdxInputData[2]!=tono){
    tono = rtdxInputData[2];
    switch (rtdxInputData[2]){
        case 0: modificadorTono = SIs_DO; break;
        case 1: modificadorTono = DOs_REb; break;
        case 2: modificadorTono = RE; break;
        case 3: modificadorTono = REs_MIb; break;
        case 4: modificadorTono = MI_FAb; break;
        case 5: modificadorTono = MIs_FA; break;
        case 6: modificadorTono = FAs_SOLb; break;
        case 7: modificadorTono = SOL; break;
        case 8: modificadorTono = SOLs_LAb; break;
        case 9: modificadorTono = LA; break;
        case 10: modificadorTono = LAs_SIb; break;
        case 11: modificadorTono = SI_DOb; break;
    }
}

//Se calcula el índice que se relaciona con la nota
frecNota = frecuenciaFundamental;
```

```
            while (frecNota>FREC_MIN){ //Se cambia el valor de la
                frecNota = frecNota/2; //frecuencia de la nota al de
            }                          //la segunda octava
            frecNota = frecNota * 2;

            indiceNota = -1; // Índice de los vectores para las
                             //distancias de terceras, quintas y
                             //séptimas

            //Se busca la nota dentro del grupo de notas de
            //frecuencia más baja empezando por la más aguda
            while (frecNota>FREC_MIN){
                indiceNota = indiceNota + 1;
                frecNota = frecNota / SEMITONO;

            }

            //Se aplican los modificadores al índice por escala y
            //tonalidad. Como el desplazamiento es circular se
            //actualiza el índice para posiciones entre 0 y 11
            indiceNota2 = indiceNota + modificadorEscala +
                          modificadorTono;
            while(indiceNota2>11){indiceNota2 = indiceNota2-12;}

            //Se determinan las distancias de cada intervalo para
            //formar el acorde
            distTercera = terceras[indiceNota2];
            distQuinta = quintas[indiceNota2];
            distSeptima = septimas[indiceNota2];
        }

/*-------------------------------------------------------------------*/
/*                          EFFECTOS                                 */
/*-------------------------------------------------------------------*/

        /* Asignar numero de notas y calcular su frecuencia */
        switch (rtdxInputData[0]){
            case 0: //bypass
                f[0] = frecuenciaFundamental;
                numNotas = 1;
                break;
            case 1: //octavador 1
                f[0] = frecuenciaFundamental;
                f[1] = frecuenciaFundamental*2;
                numNotas = 2;
                break;
            case 2: //octavador 2
                f[0] = frecuenciaFundamental;
                f[1] = frecuenciaFundamental*2;
                f[2] = frecuenciaFundamental/2;
                numNotas = 3;
                break;
            case 3: //quintas 1
                f[0] = frecuenciaFundamental;
                f[1] = frecuenciaFundamental * pow(SEMITONO, 7);
                numNotas = 2;
                break;
            case 4: //quintas 2
                f[0] = frecuenciaFundamental;
                f[1] = frecuenciaFundamental * pow(SEMITONO, 7);
                f[2] = frecuenciaFundamental * 2;
```

```
                    numNotas = 3;
                    break;



                case 5: //triada
                    if(distTercera==0){
                        if(rtdxInputData[3]==0){numNotas = 0;}
                        else {numNotas = 1;}
                    }else{
                        f[0] = frecuenciaFundamental;
                        f[1] = frecuenciaFundamental * pow(SEMITONO,
                              distTercera);
                        f[2] = frecuenciaFundamental * pow(SEMITONO,
                              distQuinta);
                        numNotas = 3;
                    }
                    break;
                case 6: //cuatríada
                    if(distTercera==0){
                        if(rtdxInputData[3]==0){numNotas = 0;}
                        else {numNotas = 1;}
                    }else{
                        f[0] = frecuenciaFundamental;
                        f[1] = frecuenciaFundamental * pow(SEMITONO,
                              distTercera);
                        f[2] = frecuenciaFundamental * pow(SEMITONO,
                              distQuinta);
                        f[3] = frecuenciaFundamental * pow(SEMITONO,
                              distSeptima);
                        numNotas = 4;
                    }
                    break;
                case 7: //diseño de armonía
                    f[0] = frecuenciaFundamental;
                    f[1] = frecuenciaFundamental * pow(SEMITONO,
                              rtdxInputData[7]);
                    f[2] = frecuenciaFundamental * pow(SEMITONO,
                              rtdxInputData[8]);
                    f[3] = frecuenciaFundamental * pow(SEMITONO,
                              rtdxInputData[9]);
                    numNotas = rtdxInputData[6] + 1;
                    break;
                default:
                    break;
            }

/*-------------------------------------------------------------------*/
/*                         DETECTOR DE NOTA                          */
/*-------------------------------------------------------------------*/

/*          Esta parte del programa aun no funciona correctamente
            Se deja para futuras mejoras


            if (rtdxInputData[10]==1){

                //Se calcula el índice que se relaciona con la nota
                frecNota = frecuenciaFundamental;
                octava = 1;
                while (frecNota>FREC_MIN){
```

```
                    frecNota = frecNota/2;
                    octava = octava + 1;
                }
                frecNota = frecNota * 2;

                indiceNota = -1;
                while (frecNota>FREC_MIN){
                   indiceNota = indiceNota + 1;
                   frecNota = frecNota / SEMITONO;
                }

                rtdxOutputData[0] = frecuenciaFundamental;
                rtdxOutputData[1] = indiceNota;
                rtdxOutputData[2] = octava;
                RTDX_write(&rtdxOutputChannel, &rtdxOutputData,
                        sizeof(rtdxOutputData));

                rtdxInputData[10] = 0;
            }*/

/*-------------------------------------------------------------------*/
/*                          CALCULAR SALIDA                          */
/*-------------------------------------------------------------------*/

            /* Generar matriz de frecuencias e incrementos */
            for(i=0; i<numNotas; i++){
                incrementoTabla[i] = (float)f[i]/frecNaturalTabla;
            }

            /* Generar señal de salida */
            for (i=0; i<PASO; i++){
                salida[i] = 0;
                for(j=0; j<numNotas; j++){
                    salida[i] = salida[i] + (amplitudEspectralMax *
                              tabla[indT[j]]);
                    // Se actualizan los índices
                    indiceTabla[j] = indiceTabla[j] +
                                     incrementoTabla[j];
                    if (indiceTabla[j]>=LONG_TABLA) {
                        indiceTabla[j] = indiceTabla[j] - LONG_TABLA;
                    }
                    indT[j] = indiceTabla[j];
                }
                salida[i] = salida[i]>>15; // Se pasa a short
            }

/*-------------------------------------------------------------------*/
/*                       BUFFERS DE SALIDA                           */
/*-------------------------------------------------------------------*/

            //Buffer a procesar en función de la variable ciclo
            if (!cicloBuffers){
                for (i=0; i<PASO; i++){    //procesado bufferIn1
                    bufferOut1[2*i] = salida[i];
                    bufferOut1[2*i+1] = salida[i];
                }
            } else {
                for (i=0; i<PASO; i++){    //procesado bufferIn2
                    bufferOut2[2*i] = salida[i];
                    bufferOut2[2*i+1] = salida[i];
                }
```

```
            }
            cicloBuffers=!cicloBuffers; //Actualiza la variable ciclo


        } //FIN del if flag interrupcion
    } //FIN del while bucle infinito
} //FIN del main


/*=====================================================================*/
/*=====================================================================*/
/*==                                                                 ==*/
/*==      RUTINAS PARA INICIALIZACIÓN DE PERIFÉRICOS                 ==*/
/*==                                                                 ==*/
/*=====================================================================*/
/*=====================================================================*/

/*---------------------------------------------------------------------*/
/*                      INICIALIZACION DEL EDMA                        */
/*---------------------------------------------------------------------*/

void edma_init(){

    //Abrir canales, reiniciarlos y deshabilitarles la interrupción
    hEdmaLeer = EDMA_open( EDMA_CHA_REVT1, EDMA_OPEN_RESET );
    hEdmaEscribir = EDMA_open( EDMA_CHA_XEVT1, EDMA_OPEN_RESET );
    hEdmaDesplazarEntradaFIR =  EDMA_open( EDMA_CHA_TCC8,
                        EDMA_OPEN_RESET ); muestrasEntradaFIR
    hEdmaActualizarEntradaFIR = EDMA_open( EDMA_CHA_TCC9,
                        EDMA_OPEN_RESET );
    hEdmaDesplazarEntrada =  EDMA_open( EDMA_CHA_EXTINT4,
                        EDMA_OPEN_RESET );
    hEdmaCopiarEntrada = EDMA_open( EDMA_CHA_TCC10, EDMA_OPEN_RESET );

    //Reservar espacio para los parámetros de recarga
    hEdmaLINKleer1 = EDMA_allocTable( -1 );
    hEdmaLINKleer2 = EDMA_allocTable( -1 );
    hEdmaLINKescribir1 = EDMA_allocTable( -1 );
    hEdmaLINKescribir2 = EDMA_allocTable( -1 );
    hEdmaLINKactualizarEntradaFIR1 = EDMA_allocTable( -1 );
    hEdmaLINKactualizarEntradaFIR2 = EDMA_allocTable( -1 );

    //Completar las variables de configuración del EDMA con las
    //direcciones de recarga correspondientes. Por eso el RELOAD se ha
    //dejado antes a 0
    edma_Config_Leer1.rld = (hEdmaLINKleer2);
    edma_Config_Leer2.rld = (hEdmaLINKleer1);
    edma_Config_Escribir1.rld = (hEdmaLINKescribir2);
    edma_Config_Escribir2.rld = (hEdmaLINKescribir1);
    edma_Config_ActualizarEntradaFIR1.rld =
      (hEdmaLINKactualizarEntradaFIR2);
     edma_Config_ActualizarEntradaFIR2.rld =
      (hEdmaLINKactualizarEntradaFIR1);

    //Asociar la configuración al canal creado
    EDMA_config(hEdmaLeer, &edma_Config_Leer1);
    EDMA_config(hEdmaEscribir, &edma_Config_Escribir1);
    EDMA_config(hEdmaDesplazarEntradaFIR,
      &edma_Config_DesplazarEntradaFIR);
    EDMA_config(hEdmaActualizarEntradaFIR,
      &edma_Config_ActualizarEntradaFIR1
```

```
    EDMA_config(hEdmaDesplazarEntrada, &edma_Config_DesplazarEntrada);
    EDMA_config(hEdmaCopiarEntrada, &edma_Config_CopiarEntrada);



    //Configurar los canales de recarga del EDMA.
    EDMA_config(hEdmaLINKleer1, &edma_Config_Leer1);
    EDMA_config(hEdmaLINKleer2, &edma_Config_Leer2);
    EDMA_config(hEdmaLINKescribir1, &edma_Config_Escribir1);
    EDMA_config(hEdmaLINKescribir2, &edma_Config_Escribir2);
    EDMA_config(hEdmaLINKactualizarEntradaFIR1,
      &edma_Config_ActualizarEntradaFIR1);
    EDMA_config(hEdmaLINKactualizarEntradaFIR2,
      &edma_Config_ActualizarEntradaFIR2);

    //Habilitar los canales
    EDMA_enableChannel( hEdmaLeer );
    EDMA_enableChannel( hEdmaEscribir );
    EDMA_enableChannel( hEdmaDesplazarEntradaFIR );
    EDMA_enableChannel( hEdmaActualizarEntradaFIR
    EDMA_enableChannel( hEdmaDesplazarEntrada );
    EDMA_enableChannel( hEdmaCopiarEntrada );

    //Borrar posibles interrupciones pendientes del EDMA.
    EDMA_intClear(4);
    EDMA_intClear(8);
    EDMA_intClear(9);
    EDMA_intClear(10);
    EDMA_intClear(12);

    //Habilitar la interrupción por finalización de lectura del EDMA.
    EDMA_intEnable(12);

    //Habilitar el chain de los canales 8, 9, 10 y 11 por finalización
    //de lectura del EDMA.
    EDMA_enableChaining(hEdmaDesplazarEntradaFIR);
    EDMA_enableChaining(hEdmaActualizarEntradaFIR);
    //EDMA_enableChaining(hEdmaDesplazarEntrada);
    EDMA_enableChaining(hEdmaCopiarEntrada);
}


/*=====================================================================*/
/*=====================================================================*/
/*==                                                                 ==*/
/*==            RUTINAS DE SERVICIO A LA INTERRUPCIÓN                ==*/
/*==                                                                 ==*/
/*=====================================================================*/
/*=====================================================================*/

/*---------------------------------------------------------------------*/
/*          RUTINA DE SERVICIO A LA INTERRUPCION DEL EDMA              */
/*---------------------------------------------------------------------*/

void EDMA_HWI(void){
      EDMA_intClear(12);      //Baja petición de interrupción
      flagEDMAinterrupt = 1;  //Activa el indicador de que se ha
                              //producido una interrupción
}
```

bitrev_index.c

```
/*===========================================================================*/
/*                                                                           */
/*  TEXAS INSTRUMENTS, INC.                                                  */
/*                                                                           */
/*  NAME                                                                     */
/*      bitrev_index.c                                                       */
/*                                                                           */
/*  USAGE                                                                    */
/*      This function has the prototype:                                     */
/*                                                                           */
/*      void bitrev_index(short *index, int n);                              */
/*                                                                           */
/*      index[sqrt(n)] : Pointer to index table that is returned             */
/*                       by the routine.                                     */
/*      n              : Number of complex array elements to                 */
/*                       bit-reverse.                                        */
/*                                                                           */
/*  DESCRIPTION                                                              */
/*      This routine calculates the index table for the DSPLIB               */
/*      function bitrev_cplx which performs a complex bit reversal           */
/*      of an array of length n. The length of the index table is            */
/*      2^(2*ceil(k/2)) where n = 2^k. In other words, the length            */
/*      of the index table is sqrt(n)for even powers of radix.               */
/*                                                                           */
/*===========================================================================*/

void bitrev_index(short *index, int n)
{
    int   i, j, k, radix = 2;
    short nbits, nbot, ntop, ndiff, n2, raddiv2;

    nbits = 0;
    i = n;
    while (i > 1)
    {
        i = i >> 1;
        nbits++;
    }

    raddiv2 = radix >> 1;
    nbot    = nbits >> raddiv2;
    nbot    = nbot << raddiv2 - 1;
    ndiff   = nbits & raddiv2;
    ntop    = nbot + ndiff;
    n2      = 1 << ntop;

    index[0] = 0;
    for ( i = 1, j = n2/radix + 1; i < n2 - 1; i++)
    {
        index[i] = j - 1;

        for (k = n2/radix; k*(radix-1) < j; k /= radix)
            j -= k*(radix-1);

        j += k;
    }
    index[n2 - 1] = n2 - 1;
}
```

tw_radix2.c

```
/*===========================================================================*/
/*  TEXAS INSTRUMENTS, INC.                                                  */
/*                                                                           */
/*  NAME                                                                     */
/*      tw_radix2.c                                                          */
/*                                                                           */
/*  USAGE                                                                    */
/*      This is a stand-alone program intended to generate                   */
/*      twiddle-factor arrays for the DSP_radix2 FFT library                 */
/*      routine.It is called as follows:                                     */
/*                                                                           */
/*                                                                           */
/*          tw_radix2 [-s scale] N > outputfile.c                            */
/*                                                                           */
/*      The argument 'N' specifies the size of the FFT.  This                */
/*      value must be a power of 2.  The switch '-s scale' allows            */
/*      selecting a different scale factor for the coefficients.             */
/*      The default scale factor is 32767.5.                                 */
/*                                                                           */
/*      This program will generate the twiddle factor array 'w'              */
/*      and output the result to the display.  Redirect the output           */
/*      to a file as shown above.                                            */
/*                                                                           */
/*  DESCRIPTION                                                              */
/*      This program contains the twiddle-factor generation routine          */
/*      that is described within the source code for the DSPLIB              */
/*      FFT function DSP_radix2.  It does not produce appropriate            */
/*      twiddle-factor arrays for the other FFT implementations.             */
/*                                                                           */
/*      Please consult the specific FFT that you are using for               */
/*      details.                                                             */
/*                                                                           */
/*      The final twiddle-factor array generated by the routine              */
/*      will be 2 * N elements long.                                         */
/*                                                                           */
/*  NOTES                                                                    */
/*      The code below may be adapted to run directly on the                 */
/*      target, rather than running as a separate program running            */
/*      off-line.                                                            */
/*      Such adaptation is not recommended for time-critical                 */
/*      applications.                                                        */
/*      To adapt this program, remove the 'usage' and 'main'                 */
/*      functions, and call 'gen_twiddle' directly.                          */
/*                                                                           */
/* ------------------------------------------------------------------------- */
/*        Copyright (c) 2001 Texas Instruments, Incorporated.                */
/*                       All Rights Reserved.                                */
/*===========================================================================*/
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#ifndef PI
# ifdef M_PI
#  define PI M_PI
# else
#  define PI 3.14159265358979323846
```

```
# endif
#endif

/*=========================================================================*/
/*  D2S -- Truncate a 'double' to a 'short', with clamping.                */
/*=========================================================================*/
static short d2s(double d)
{
    if (d >=  32767.0) return  32767;
    if (d <= -32768.0) return -32768;
    return (short)d;
}

/*=========================================================================*/
/* GEN_TWIDDLE -- Generate twiddle factors for TI's custom FFTs.           */
/*                                                                         */
/*  USAGE                                                                  */
/*      This routine is called as follows:                                 */
/*                                                                         */
/*          int gen_twiddle(short *w, int n, double scale)                 */
/*                                                                         */
/*          short  *w     Pointer to twiddle-factor array                  */
/*          int    n      Size of FFT                                      */
/*          double scale  Scale factor to apply to values.                 */
/*                                                                         */
/*      The routine will generate the twiddle-factors directly into        */
/*      the array you specify.  The array needs to be N elements           */
/*      long.                                                              */
/*=========================================================================*/
int gen_twiddle(short *w, int n, double scale)
{
    double angle, step;
    int i;

    step = (2.0 * PI) / (double)n;
    for (i = 0, angle = 0.0; i < n; i += 2, angle += step)
    {
        w[i + 0] = d2s(scale * -cos(angle));
        w[i + 1] = d2s(scale * -sin(angle));
    }

    return n;
}
```

armonizador.h

```
// AUTOR: Carlos Anchuela Arnalte.
// PROYECTO: Armonizador para instrumento musical.
// NOMBRE DEL ARCHIVO: armonizador.h

/*====================================================================*/
/*====================================================================*/
/*==                                                                ==*/
/*==                    PROTOTIPOS DE FUNCIONES                     ==*/
/*==                                                                ==*/
/*====================================================================*/
/*====================================================================*/

void edma_init(void);
void EDMA_HWI(void);
```

bitrev_index.h

```
/*====================================================================*/
/*                                                                    */
/*  TEXAS INSTRUMENTS, INC.                                           */
/*                                                                    */
/*  NAME                                                              */
/*      bitrev_index.h                                                */
/*                                                                    */
/*====================================================================*/

void bitrev_index(short *index, int n);
```

tw_radix2.h

```
/*====================================================================*/
/*  TEXAS INSTRUMENTS, INC.                                           */
/*                                                                    */
/*  NAME                                                              */
/*      tw_radix2.h                                                   */
/*                                                                    */
/* ------------------------------------------------------------------ */
/*        Copyright (c) 2001 Texas Instruments, Incorporated.         */
/*                        All Rights Reserved.                        */
/*====================================================================*/

static short d2s(double d);
int gen_twiddle(short *w, int n, double scale);
```

acordes.h

```
// AUTOR: Carlos Anchuela Arnalte.
// PROYECTO: Armonizador para instrumento musical.
// NOMBRE DEL ARCHIVO: acordes.h

/*====================================================================*/
/*====================================================================*/
/*==                                                                ==*/
/*==             ACORDES SEGUN ESCALAS Y TONALIDAD                  ==*/
/*==                                                                ==*/
/*====================================================================*/
/*====================================================================*/

// MODIFICADOR POR MODO/ESCALA

#define MAYOR          0
#define MENOR_NATURAL  9
#define MENOR_ARMONICA 0
#define MENOR_MELODICA 0
#define JONICO         0
#define DORICO         2
#define FRIGIO         4
#define LIDIO          5
#define MIXOLIDIO      7
#define EOLICO         9
#define LOCRIO        11

// MODIFICADOR POR TONALIDAD

#define SIs_DO    0
#define DOs_REb  11
#define RE       10
#define REs_MIb   9
#define MI_FAb    8
#define MIs_FA    7
#define FAs_SOLb  6
#define SOL       5
#define SOLs_LAb  4
#define LA        3
#define LAs_SIb   2
#define SI_DOb    1
```

```
// TABLAS DE DISTANCIAS EN SEMITONOS PARA FORMAR ACORDES

//     Si#  Reb       Mib  Fab  Mi#  Solb      Lab       Sib  Dob
//     Do   Do#  Re   Re#  Mi   Fa   Fa#  Sol  Sol#  La  La#  Si
//-------------------------------------------------------------------
//      0    1    2    3    4    5    6    7    8    9   10   11
//-------------------------------------------------------------------
         short base3[12] =
      { 4,   0,   3,   0,   3,   4,   0,   4,   0,   3,   0,   3};
         short base5[12] =
      { 7,   0,   7,   0,   7,   7,   0,   7,   0,   7,   0,   6};
         short base7[12] =
      {11,   0,  10,   0,  10,  11,   0,  10,   0,  10,   0,  10};
//-------------------------------------------------------------------
short menorArmonica3[12] =
      { 3,   0,   3,   4,   0,   3,   0,   4,   4,   0,   0,   3};
short menorArmonica5[12] =
      { 7,   0,   6,   8,   0,   7,   0,   7,   7,   0,   0,   6};
short menorArmonica7[12] =
      {11,   0,  10,  11,   0,  10,   0,  10,  11,   0,   0,   9};
//-------------------------------------------------------------------
short menorMelodica3[12] =
      { 3,   0,   3,   4,   0,   4,   0,   4,   0,   3,   0,   3};
short menorMelodica5[12] =
      { 7,   0,   7,   8,   0,   7,   0,   7,   0,   6,   0,   6};
short menorMelodica7[12] =
      {10,   0,  10,  11,   0,  10,   0,  10,   0,  10,   0,  10};
```

FIR_41.h

```
// AUTOR: Carlos Anchuela Arnalte.
// PROYECTO: Armonizador para instrumento musical.
// NOMBRE DEL ARCHIVO: FIR_41.h

/*
 * Filter Coefficients (C Source) generated by the Filter Design and
 * Analysis Tool
 *
 * Generated by MATLAB(R) 7.8 and the Filter Design Toolbox 4.5.
 *
 * Generated on: 21-Jul-2013 23:33:36
 *
 */

/*
 * Discrete-Time FIR Filter (real)
 * -------------------------------
 * Filter Structure  : Direct-Form FIR
 * Filter Length     : 41
 * Stable            : Yes
 * Linear Phase      : Yes (Type 1)
 * Arithmetic        : fixed
 * Numerator         : s16,16 -> [-5.000000e-001 5.000000e-001)
 * Input             : s16,15 -> [-1 1)
 * Filter Internals  : Full Precision
 *   Output          : s33,31 -> [-2 2)  (auto determined)
 *   Product         : s31,31 -> [-5.000000e-001 5.000000e-001)  (auto
 *                                determined)
 *   Accumulator     : s33,31 -> [-2 2)  (auto determined)
 *   Round Mode      : No rounding
 *   Overflow Mode   : No overflow
 */

/* General type conversion for MATLAB generated C-code  */

/*
 * Expected path to tmwtypes.h
 * C:\Archivos de programa\MATLAB\R2009a\extern\include\tmwtypes.h
 */

short coefFIR[41] = {
       -2,    13,   495,  1020,  1132,   384,  -711,  -888,   274,
     1320,   509, -1460, -1683,   941,  3086,   705, -4433, -4560,
     5407, 20005, 27005, 20005,  5407, -4560, -4433,   705,  3086,
      941, -1683, -1460,   509,  1320,   274,  -888,  -711,   384,
     1132,  1020,   495,   113,    -2
};
```

hamming.h

```
// AUTOR: Carlos Anchuela Arnalte.
// PROYECTO: Armonizador para instrumento musical.
// NOMBRE DEL ARCHIVO: hamming.h

unsigned short coefVentana[1024] = {

2621,2622,2623,2624,2626,2629,2632,2635,2640,2644,2650,2656,2662,2669,
2677,2685,2694,2704,2713,2724,2735,2747,2759,2772,2785,2799,2813,2828,
2844,2860,2877,2894,2912,2930,2949,2968,2988,3009,3030,3052,3074,3097,
3120,3144,3169,3194,3219,3245,3272,3299,3327,3355,3384,3413,3443,3473,
3504,3536,3568,3600,3633,3667,3701,3736,3771,3807,3843,3880,3917,3955,
3993,4032,4071,4111,4152,4193,4234,4276,4318,4361,4405,4449,4493,4538,
4583,4629,4676,4723,4770,4818,4866,4915,4964,5014,5065,5115,5167,5218,
5270,5323,5376,5430,5484,5538,5593,5649,5705,5761,5818,5875,5933,5991,
6049,6108,6168,6228,6288,6349,6410,6471,6533,6596,6659,6722,6786,6850,
6914,6979,7044,7110,7176,7243,7310,7377,7445,7513,7581,7650,7719,7789,
7859,7929,8000,8071,8142,8214,8286,8359,8432,8505,8578,8652,8727,8801,
8876,8951,9027,9103,9179,9256,9332,9410,9487,9565,9643,9721,9800,9879,
9959,10038,10118,10198,10279,10359,10440,10522,10603,10685,10767,10850
,10932,11015,11098,11181,11265,11349,11433,11517,11602,11687,11772,118
57,11942,12028,12114,12200,12286,12373,12460,12547,12634,12721,12808,1
2896,12984,13072,13160,13249,13337,13426,13515,13604,13693,13782,13872
,13961,14051,14141,14231,14321,14412,14502,14593,14683,14774,14865,149
56,15047,15138,15229,15321,15412,15504,15595,15687,15779,15871,15963,1
6055,16147,16239,16331,16423,16516,16608,16700,16793,16885,16978,17070
,17163,17255,17348,17440,17533,17625,17718,17810,17903,17996,18088,181
81,18273,18366,18458,18551,18643,18735,18828,18920,19012,19104,19197,1
9289,19381,19473,19565,19656,19748,19840,19931,20023,20114,20206,20297
,20388,20479,20570,20661,20752,20842,20933,21023,21113,21203,21293,213
83,21473,21562,21652,21741,21830,21919,22008,22097,22185,22273,22361,2
2449,22537,22625,22712,22799,22886,22973,23060,23146,23232,23318,23404
,23490,23575,23660,23745,23830,23914,23998,24082,24166,24250,24333,244
16,24499,24581,24663,24745,24827,24908,24990,25070,25151,25231,25311,2
5391,25471,25550,25629,25707,25785,25863,25941,26018,26095,26172,26249
,26325,26400,26476,26551,26626,26700,26774,26848,26921,26994,27067,271
39,27211,27283,27354,27425,27495,27566,27635,27705,27774,27842,27911,2
7979,28046,28113,28180,28246,28312,28378,28443,28507,28572,28636,28699
,28762,28825,28887,28949,29010,29071,29132,29192,29251,29311,29369,294
28,29486,29543,29600,29657,29713,29768,29824,29878,29933,29986,30040,3
0093,30145,30197,30249,30300,30350,30400,30450,30499,30547,30596,30643
,30690,30737,30783,30829,30874,30919,30963,31007,31050,31092,31135,311
76,31217,31258,31298,31338,31377,31415,31454,31491,31528,31565,31601,3
1636,31671,31705,31739,31773,31805,31838,31869,31901,31931,31962,31991
,32020,32049,32077,32104,32131,32157,32183,32208,32233,32257,32281,323
04,32327,32349,32370,32391,32411,32431,32450,32469,32487,32504,32521,3
2538,32554,32569,32584,32598,32611,32624,32637,32649,32660,32671,32681
,32691,32700,32708,32716,32724,32730,32737,32742,32747,32752,32756,327
59,32762,32765,32766,32767,32768,32768,32767,32766,32765,32762,32759,3
2756,32752,32747,32742,32737,32730,32724,32716,32708,32700,32691,32681
,32671,32660,32649,32637,32624,32611,32598,32584,32569,32554,32538,325
21,32504,32487,32469,32450,32431,32411,32391,32370,32349,32327,32304,3
2281,32257,32233,32208,32183,32157,32131,32104,32077,32049,32020,31991
,31962,31931,31901,31869,31838,31805,31773,31739,31705,31671,31636,316
01,31565,31528,31491,31454,31415,31377,31338,31298,31258,31217,31176,3
1135,31092,31050,31007,30963,30919,30874,30829,30783,30737,30690,30643
,30596,30547,30499,30450,30400,30350,30300,30249,30197,30145,30093,300
40,29986,29933,29878,29824,29768,29713,29657,29600,29543,29486,29428,2
9369,29311,29251,29192,29132,29071,29010,28949,28887,28825,28762,28699
,28636,28572,28507,28443,28378,28312,28246,28180,28113,28046,27979,279
```

```
11,27842,27774,27705,27635,27566,27495,27425,27354,27283,27211,27139,2
7067,26994,26921,26848,26774,26700,26626,26551,26476,26400,26325,26249
,26172,26095,26018,25941,25863,25785,25707,25629,25550,25471,25391,253
11,25231,25151,25070,24990,24908,24827,24745,24663,24581,24499,24416,2
4333,24250,24166,24082,23998,23914,23830,23745,23660,23575,23490,23404
,23318,23232,23146,23060,22973,22886,22799,22712,22625,22537,22449,223
61,22273,22185,22097,22008,21919,21830,21741,21652,21562,21473,21383,2
1293,21203,21113,21023,20933,20842,20752,20661,20570,20479,20388,20297
,20206,20114,20023,19931,19840,19748,19656,19565,19473,19381,19289,191
97,19104,19012,18920,18828,18735,18643,18551,18458,18366,18273,18181,1
8088,17996,17903,17810,17718,17625,17533,17440,17348,17255,17163,17070
,16978,16885,16793,16700,16608,16516,16423,16331,16239,16147,16055,159
63,15871,15779,15687,15595,15504,15412,15321,15229,15138,15047,14956,1
4865,14774,14683,14593,14502,14412,14321,14231,14141,14051,13961,13872
,13782,13693,13604,13515,13426,13337,13249,13160,13072,12984,12896,128
08,12721,12634,12547,12460,12373,12286,12200,12114,12028,11942,11857,1
1772,11687,11602,11517,11433,11349,11265,11181,11098,11015,10932,10850
,10767,10685,10603,10522,10440,10359,10279,10198,10118,10038,9959,9879
,9800,9721,9643,9565,9487,9410,9332,9256,9179,9103,9027,8951,8876,8801
,8727,8652,8578,8505,8432,8359,8286,8214,8142,8071,8000,7929,7859,7789
,7719,7650,7581,7513,7445,7377,7310,7243,7176,7110,7044,6979,6914,6850
,6786,6722,6659,6596,6533,6471,6410,6349,6288,6228,6168,6108,6049,5991
,5933,5875,5818,5761,5705,5649,5593,5538,5484,5430,5376,5323,5270,5218
,5167,5115,5065,5014,4964,4915,4866,4818,4770,4723,4676,4629,4583,4538
,4493,4449,4405,4361,4318,4276,4234,4193,4152,4111,4071,4032,3993,3955
,3917,3880,3843,3807,3771,3736,3701,3667,3633,3600,3568,3536,3504,3473
,3443,3413,3384,3355,3327,3299,3272,3245,3219,3194,3169,3144,3120,3097
,3074,3052,3030,3009,2988,2968,2949,2930,2912,2894,2877,2860,2844,2828
,2813,2799,2785,2772,2759,2747,2735,2724,2713,2704,2694,2685,2677,2669
,2662,2656,2650,2644,2640,2635,2632,2629,2626,2624,2623,2622,2621

};
```

dsp_bitrev_cplx.h

```
/*=========================================================================*/
/*=========================================================================*/
/*  TEXAS INSTRUMENTS, INC.                                                */
/*                                                                         */
/*  NAME                                                                   */
/*      DSP_bitrev_cplx.h                                                  */
/*                                                                         */
/*  REVISION DATE                                                          */
/*      18-Sep-2001                                                        */
/*                                                                         */
/*  USAGE                                                                  */
/*      This routine is C Callable and can be called as:                   */
/*                                                                         */
/*      void DSP_bitrev_cplx(int *x, short *index, int nx);                */
/*                                                                         */
/*      x[nx]  : Complex input array to be bit-reversed. One element*/
/*               consists of a pair of 16-bit data.                        */
/*      index[]: Array of size ~sqrt(nx) created by the routine            */
/*               bitrev_index to allow the fast implementation of          */
/*               the bit-reversal.                                         */
/*      nx     : Number of elements in array x[]. Must be power of 2*/
/*                                                                         */
/*  DESCRIPTION                                                            */
/*      This routine performs the bit-reversal of the input array          */
/*      x[],where x[] is an integer array of length nx containing          */
/*      16-bit complex pairs of data.  This routine requires the           */
/*      index array provided by the program below.  This index             */
/*      should be generated at compile time not by the DSP.                */
/*                                                                         */
/*      authorizes the use of the bit-reversal code and related            */
/*      table generation code with TMS320-family DSPs manufactured         */
/*      by TI.                                                             */
/*  ASSUMPTIONS                                                            */
/*      nx must be a power of 2.                                           */
/*      The table from bitrev_index is already created.                    */
/*      LITTLE ENDIAN configuration used.                                  */
/*  NOTES                                                                  */
/*      If nx <= 4K one can use the char (8-bit) data type for             */
/*      the "index" variable. This would require changing the LDH          */
/*      when loading index values in the assembly routine to LDB.          */
/*      This would further reduce the size of the Index Table by           */
/*      half its size.                                                     */
/*      This code is interrupt tolerant, but not interruptible.            */
/*  CYCLES                                                                 */
/*      (nx/4 + 2) * 7 + 18                                                */
/*      e.g. nx = 256, cycles = 480                                        */
/*  CODESIZE                                                               */
/*      352 bytes                                                          */
/*                                                                         */
/* ----------------------------------------------------------------------- */
/*        Copyright (c) 2003 Texas Instruments, Incorporated.              */
/*                       All Rights Reserved.                              */
/*=========================================================================*/
#ifndef DSP_BITREV_CPLX_H_
#define DSP_BITREV_CPLX_H_ 1

void DSP_bitrev_cplx(int *x, short *index, int nx);

#endif
```

dsp_radix2.h

```
/*===========================================================================*/
/*                                                                           */
/*  TEXAS INSTRUMENTS, INC.                                                  */
/*                                                                           */
/*  NAME                                                                     */
/*      DSP_radix2.c                                                         */
/*                                                                           */
/*  REVISION DATE                                                            */
/*      09-Dec-2002                                                          */
/*  USAGE                                                                    */
/*      This routine is C-callable and can be called as:                     */
/*                                                                           */
/*      void DSP_radix2(int n, short *restrict xy,                           */
/*                   const short *restrict w);                               */
/*      n    -- FFT size                                 (input)             */
/*      xy[] -- input and output sequences (dim-n)  (input/output)  */
/*      w[]  -- FFT coefficients (dim-n/2)          (input)             */
/*                                                                           */
/*  DESCRIPTION                                                              */
/*      This routine is used to compute FFT of a complex sequece of */
/*      size n, a power of 2, with "decimation-in-frequency         */
/*      decomposition"method, ie, the output is in bit-reversed     */
/*      order.Each complex value is with interleaved 16-bit real and*/
/*      imaginary parts. To prevent overflow, input samples may have*/
/*      to be scaled by 1/n.                                                 */
/*  ASSUMPTIONS                                                              */
/*      16 <= n <= 32768                                                     */
/*      Both input xy and coefficient w must be aligned on word      */
/*      boundary.w coef stored ordered is k*(-cos[0*delta]), k       */
/*      *(-sin[0*delta]),k*(-cos[1*delta]), ...  where               */
/*      delta = 2*PI/N, k = 32767                                            */
/*      Assembly code is written for processor in Little Endian mode*/
/*      Input xy and coefficients w are 16 bit data.                 */
/*  MEMORY NOTE                                                              */
/*      Align xy and w on different word boundaries to minimize      */
/*      memory bank hits.                                                    */
/*  TECHNIQUES                                                               */
/*      1. Loading input xy as well as coefficient w in word.        */
/*      2. Both loops j and i shown in the C code are placed in the */
/*         INNERLOOP of the assembly code.                           */
/*  CYCLES                                                                   */
/*      cycles = log2(N) * (4*N/2+7) + 34 + N/4.                     */
/*      (The N/4 term is due to bank conflicts that occur when xy    */
/*      and w are aligned as suggested above, under "MEMORY NOTE.") */
/*      For N = 256, cycles = 4250.                                  */
/*  CODESIZE                                                                 */
/*      800 bytes                                                            */
/*                                                                           */
/* ------------------------------------------------------------------------- */
/*        Copyright (c) 2003 Texas Instruments, Incorporated.        */
/*                        All Rights Reserved.                       */
/*===========================================================================*/
#ifndef DSP_RADIX2_H_
#define DSP_RADIX2_H_ 1

void DSP_radix2(int n, short *restrict xy,
            const short *restrict w);

#endif
```

Apéndice E. Código Visual C++

Visual C++ genera gran cantidad de código automáticamente para el mantenimiento de las clases y funciones. Para diferenciar el código que genera Visual C++ con el que se ha desarrollado, se ha indicado el código de Visual C++ en color gris.

armonizadorDlg.cpp

```
// armonizadorDlg.cpp : implementation file
//

#include <windows.h>

#include "stdafx.h"
#include "armonizador.h"
#include "armonizadorDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
```

```
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CArmonizadorDlg dialog

CArmonizadorDlg::CArmonizadorDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CArmonizadorDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CArmonizadorDlg)
	m_efecto = 0;
	m_escala = 0;
	m_tono = 0;
	m_volumen = 48;
	m_gainIn = 0;
	m_notaNoEscala = FALSE;
	m_intervaloN = 1;
	m_intervalo1 = -12;
	m_intervalo2 = 7;
	m_intervalo3 = 12;
	m_textFrecuencia = 0.0f;
	m_textNota = _T("");
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon
in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CArmonizadorDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CArmonizadorDlg)
	DDX_Control(pDX, IDC_BUTTON_DET_NOTA, m_detectarNota);
	DDX_Control(pDX, IDC_BUTTON_ARMONIA, m_aceptarArmonia);
	DDX_Radio(pDX, IDC_EF_BYPASS, m_efecto);
	DDX_Radio(pDX, IDC_ESC_MAYOR, m_escala);
	DDX_Radio(pDX, IDC_TONO_DO, m_tono);
	DDX_Slider(pDX, IDC_SLIDER_VOLUMEN, m_volumen);
	DDX_Slider(pDX, IDC_SLIDER_GAIN_IN, m_gainIn);
	DDX_Check(pDX, IDC_CHECK_NOTA_NO_ESCALA, m_notaNoEscala);
	DDX_Text(pDX, IDC_EDIT_INTERVALO_N, m_intervaloN);
	DDV_MinMaxInt(pDX, m_intervaloN, 1, 3);
	DDX_Text(pDX, IDC_EDIT_INTERVALO1, m_intervalo1);
	DDV_MinMaxInt(pDX, m_intervalo1, -12, 12);
	DDX_Text(pDX, IDC_EDIT_INTERVALO2, m_intervalo2);
	DDV_MinMaxInt(pDX, m_intervalo2, -12, 12);
	DDX_Text(pDX, IDC_EDIT_INTERVALO3, m_intervalo3);
	DDV_MinMaxInt(pDX, m_intervalo3, -12, 12);
	DDX_Text(pDX, IDC_EDIT_DET_FRECUENCIA, m_textFrecuencia);
	DDV_MinMaxFloat(pDX, m_textFrecuencia, 0.f, 2000.f);
	DDX_Text(pDX, IDC_EDIT_DET_NOTA, m_textNota);
	DDV_MaxChars(pDX, m_textNota, 10);
	//}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CArmonizadorDlg, CDialog)
      //{{AFX_MSG_MAP(CArmonizadorDlg)
      ON_WM_SYSCOMMAND()
      ON_WM_PAINT()
      ON_WM_QUERYDRAGICON()
      ON_BN_CLICKED(IDC_CHECK_NOTA_NO_ESCALA, OnCheckNotaNoEscala)
      ON_BN_CLICKED(IDC_EF_BYPASS, OnEfBypass)
      ON_BN_CLICKED(IDC_EF_OCTAVADOR1, OnEfOctavador1)
      ON_BN_CLICKED(IDC_EF_OCTAVADOR2, OnEfOctavador2)
      ON_BN_CLICKED(IDC_EF_QUINTAS1, OnEfQuintas1)
      ON_BN_CLICKED(IDC_EF_QUINTAS2, OnEfQuintas2)
      ON_BN_CLICKED(IDC_EF_TRIADA, OnEfTriada)
      ON_BN_CLICKED(IDC_EF_CUATRIADA, OnEfCuatriada)
      ON_BN_CLICKED(IDC_EF_ARMONIA, OnEfArmonia)
      ON_BN_CLICKED(IDC_ESC_MAYOR, OnEscMayor)
      ON_BN_CLICKED(IDC_ESC_MENOR_NATURAL, OnEscMenorNatural)
      ON_BN_CLICKED(IDC_ESC_MENOR_ARMONICA, OnEscMenorArmonica)
      ON_BN_CLICKED(IDC_ESC_MENOR_MELODICA, OnEscMenorMelodica)
      ON_BN_CLICKED(IDC_ESC_JONICO, OnEscJonico)
      ON_BN_CLICKED(IDC_ESC_DORICO, OnEscDorico)
      ON_BN_CLICKED(IDC_ESC_FRIGIO, OnEscFrigio)
      ON_BN_CLICKED(IDC_ESC_LIDIO, OnEscLidio)
      ON_BN_CLICKED(IDC_ESC_MIXOLIDIO, OnEscMixolidio)
      ON_BN_CLICKED(IDC_ESC_EOLICO, OnEscEolico)
      ON_BN_CLICKED(IDC_ESC_LOCRIO, OnEscLocrio)
      ON_NOTIFY(NM_RELEASEDCAPTURE, IDC_SLIDER_GAIN_IN,
                  OnReleasedcaptureSliderGainIn)
      ON_NOTIFY(NM_RELEASEDCAPTURE, IDC_SLIDER_VOLUMEN,
                  OnReleasedcaptureSliderVolumen)
      ON_BN_CLICKED(IDC_TONO_DO, OnTonoDo)
      ON_BN_CLICKED(IDC_TONO_DO_B, OnTonoDoB)
      ON_BN_CLICKED(IDC_TONO_DO_S, OnTonoDoS)
      ON_BN_CLICKED(IDC_TONO_RE, OnTonoRe)
      ON_BN_CLICKED(IDC_TONO_RE_B, OnTonoReB)
      ON_BN_CLICKED(IDC_TONO_RE_S, OnTonoReS)
      ON_BN_CLICKED(IDC_TONO_MI, OnTonoMi)
      ON_BN_CLICKED(IDC_TONO_MI_B, OnTonoMiB)
      ON_BN_CLICKED(IDC_TONO_MI_S, OnTonoMiS)
      ON_BN_CLICKED(IDC_TONO_FA, OnTonoFa)
      ON_BN_CLICKED(IDC_TONO_FA_B, OnTonoFaB)
      ON_BN_CLICKED(IDC_TONO_FA_S, OnTonoFaS)
      ON_BN_CLICKED(IDC_TONO_SOL, OnTonoSol)
      ON_BN_CLICKED(IDC_TONO_SOL_B, OnTonoSolB)
      ON_BN_CLICKED(IDC_TONO_SOL_S, OnTonoSolS)
      ON_BN_CLICKED(IDC_TONO_LA, OnTonoLa)
      ON_BN_CLICKED(IDC_TONO_LA_B, OnTonoLaB)
      ON_BN_CLICKED(IDC_TONO_LA_S, OnTonoLaS)
      ON_BN_CLICKED(IDC_TONO_SI, OnTonoSi)
      ON_BN_CLICKED(IDC_TONO_SI_B, OnTonoSiB)
      ON_BN_CLICKED(IDC_TONO_SI_S, OnTonoSiS)
      ON_EN_CHANGE(IDC_EDIT_INTERVALO_N, OnChangeEditIntervaloN)
      ON_EN_KILLFOCUS(IDC_EDIT_INTERVALO_N, OnKillfocusEditIntervaloN)
      ON_EN_CHANGE(IDC_EDIT_INTERVALO1, OnChangeEditIntervalo1)
      ON_EN_KILLFOCUS(IDC_EDIT_INTERVALO1, OnKillfocusEditIntervalo1)
      ON_EN_CHANGE(IDC_EDIT_INTERVALO2, OnChangeEditIntervalo2)
      ON_EN_KILLFOCUS(IDC_EDIT_INTERVALO2, OnKillfocusEditIntervalo2)
      ON_EN_CHANGE(IDC_EDIT_INTERVALO3, OnChangeEditIntervalo3)
      ON_EN_KILLFOCUS(IDC_EDIT_INTERVALO3, OnKillfocusEditIntervalo3)
      ON_BN_CLICKED(IDC_BUTTON_ARMONIA, OnButtonArmonia)
      ON_BN_CLICKED(IDC_BUTTON_DET_NOTA, OnButtonDetNota)
```

```
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CArmonizadorDlg message handlers

BOOL CArmonizadorDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
					strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this
	// automatically when the application's main window is not a
	// dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon


	//************************************************************
	//                      INICIALIZACIONES
	//************************************************************

	// INICIALIZACION DE VARIABLES:

	// arrays para la trama de datos:
	for (int i=0; i<LONG_RTDX; i++) {rtdxInputData[i] = 0;}
	for (int j=0; j<LONG_RTDX; j++) {rtdxOutputData[j] = 0;}

	rtdxOutputData[0] = 0;  // Efecto Bypass
	rtdxOutputData[1] = 0;  // Escala Mayor
	rtdxOutputData[2] = 0;  // Tonalidad de Do
	rtdxOutputData[3] = 0;  // Nota fuera de escala OFF
	rtdxOutputData[4] = 0;  // Sin ganancia de entrada (la muestras
	                        // de entrada se dividen por VENTANA)
	rtdxOutputData[5] = 48; // Volumen normal (la amplitud se
	                        // multiplica por 1.012)
	rtdxOutputData[6] = 1;  // Numero de intervalos
	rtdxOutputData[7] = -12;// Distancia del 1er intervalo
	rtdxOutputData[8] = 7;  // Distancia del 2do intervalo
	rtdxOutputData[9] = 12; // Distancia del 3er intervalo

	a=0;
```

```
	// INICIALIZACION DE VENTANA:

	OnDesactivarAcorde();
	OnDesactivarArmonia();


	// INICIALIZACION COMUNICACION RTDX:

	// Escribir datos
	w_RTDX = new IRtdxExp;
	w_RTDX->CreateDispatch(_T("RTDX"));
	if(!w_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
	    MessageBox("Imposible inicializar el procesador","Error");

	// Leer datos
	r_RTDX = new IRtdxExp;
	r_RTDX->CreateDispatch(_T("RTDX"));
	if(!r_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
	    MessageBox("Imposible inicializar el procesador","Error");

	// Configuración del RTDX para la recepción de datos:
	// Modo continuo,4 buffers de 1024
	w_RTDX->DisableRtdx();
	w_RTDX->ConfigureRtdx(1,1024,4);
	w_RTDX->EnableRtdx();

	// Mira si el canal de escritura (W) está abierto
	if(w_RTDX->Open("rtdxInputChannel","W"))
	   MessageBox("No se puede abrir el canal","Error");

	// Mira si el canal de lectura (R) está abierto
	if(r_RTDX->Open("rtdxOutputChannel","R"))
	   MessageBox("No se puede abrir el canal","Error");

	//************************************************************

	return TRUE;  // return TRUE  unless you set the focus to a
	              //control
}

void CArmonizadorDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID & 0xFFF0) == IDM_ABOUTBOX)
	{
		CAboutDlg dlgAbout;
		dlgAbout.DoModal();
	}
	else
	{
		CDialog::OnSysCommand(nID, lParam);
	}
}

// If you add a minimize button to your dialog, you will need the code
below
//  to draw the icon.  For MFC applications using the document/view
model,
//  this is automatically done for you by the framework.
```

```
void CArmonizadorDlg::OnPaint()
{
      if (IsIconic())
      {
            CPaintDC dc(this); // device context for painting

            SendMessage(WM_ICONERASEBKGND,
                  (WPARAM) dc.GetSafeHdc(), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;

            // Draw the icon
            dc.DrawIcon(x, y, m_hIcon);
      }
      else
      {
            CDialog::OnPaint();
      }
}
// The system calls this to obtain the cursor to display while the
user drags
//  the minimized window.
HCURSOR CArmonizadorDlg::OnQueryDragIcon()
{
      return (HCURSOR) m_hIcon;
}

//*********************************************************************
//        RUTINA DE COMUNICACION PARA ENVIAR DATOS A LA DSP
//*********************************************************************

void CArmonizadorDlg::OnEnviarDatos()
{
    VARIANT sa;                    // Declaración del objeto
    SAFEARRAYBOUND rgsabound[1]; // Dimensión del objeto
    ::VariantInit(&sa);           // Inicialización del objeto
    sa.vt = VT_ARRAY | VT_I4;     // Objeto de enteros de 4 bytes
    rgsabound[0].lLbound = 0;     // Valor mínimo del incremento
    rgsabound[0].cElements = LONG_RTDX; // Nº elementos igual al
                                        // tamaño del array de
                                        // comunicaciones
    // Se crea el objeto con la configuración anterior:
    sa.parray = SafeArrayCreate(VT_I4, 1, rgsabound);
    // Se pasan al objeto los valores del array de comunicación:
    HRESULT hr;
    for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++)
    {
        hr = ::SafeArrayPutElement(sa.parray, &i,
                                   (long*)&rtdxOutputData[i]);
    }
    long bufferstate;
    w_RTDX->Write(sa, &bufferstate); //Se envía el objeto al buffer
    ::VariantClear(&sa); // Se limpia el valor del objeto
    w_RTDX->Flush();      // Se envían los datos al DSP
}
```

```
//*********************************************************************
//       RUTINA DE COMUNICACION PARA RECIBIR DATOS DESDE LA DSP
//*********************************************************************

void CArmonizadorDlg::OnRecibirDatos()
{
    VARIANT sa;                   // Declaración del objeto
    SAFEARRAYBOUND rgsabound[1]; // Dimensión del objeto
    ::VariantInit(&sa);          // Inicialización del objeto
    sa.vt = VT_ARRAY | VT_R4;    // Objeto de floats de 4 bytes
    rgsabound[0].lLbound = 0;    // Valor mínimo del incremento
    rgsabound[0].cElements = LONG_RTDX; // Nº elementos igual al
                                        // tamaño del array de
                                        // comunicaciones
    // Se crea el objeto con la configuración anterior:
    sa.parray = SafeArrayCreate(VT_R4, 1, rgsabound);
    r_RTDX->ReadSAF4(&sa);
    r_RTDX->Rewind();
    // Se pasan a la array de comunicaciones los valores del objeto:
    HRESULT hr;
    for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++)
    {
        hr = ::SafeArrayGetElement(sa.parray, &i,
                                   &rtdxInputData[i]);
    }
    ::VariantClear(&sa); // Se limpia el valor del objeto
}

//*********************************************************************
//                        RUTINAS DE VENTANA
//*********************************************************************

// ACORDE:
void CArmonizadorDlg::OnDesactivarAcorde()
{
      //GetDlgItem(IDC_BOX_ACORDE)->EnableWindow(FALSE);
      GetDlgItem(IDC_BOX_ESCALA)->EnableWindow(FALSE);
      GetDlgItem(IDC_BOX_MODO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_MAYOR)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_MENOR_NATURAL)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_MENOR_ARMONICA)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_MENOR_MELODICA)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_JONICO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_DORICO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_FRIGIO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_LIDIO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_MIXOLIDIO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_EOLICO)->EnableWindow(FALSE);
      GetDlgItem(IDC_ESC_LOCRIO)->EnableWindow(FALSE);
      OnDesactivarTonos();
      GetDlgItem(IDC_BOX_NOTA_NO_ESCALA)->EnableWindow(FALSE);
      GetDlgItem(IDC_CHECK_NOTA_NO_ESCALA)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnActivarAcorde()
{
      //GetDlgItem(IDC_BOX_ACORDE)->EnableWindow(TRUE);
      GetDlgItem(IDC_BOX_ESCALA)->EnableWindow(TRUE);
      GetDlgItem(IDC_BOX_MODO)->EnableWindow(TRUE);
      GetDlgItem(IDC_ESC_MAYOR)->EnableWindow(TRUE);
      GetDlgItem(IDC_ESC_MENOR_NATURAL)->EnableWindow(TRUE);
      GetDlgItem(IDC_ESC_MENOR_ARMONICA)->EnableWindow(TRUE);
```

```
	GetDlgItem(IDC_ESC_MENOR_MELODICA)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_JONICO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_DORICO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_FRIGIO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_LIDIO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_MIXOLIDIO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_EOLICO)->EnableWindow(TRUE);
	GetDlgItem(IDC_ESC_LOCRIO)->EnableWindow(TRUE);
	OnActivarTonos();
	GetDlgItem(IDC_BOX_NOTA_NO_ESCALA)->EnableWindow(TRUE);
	GetDlgItem(IDC_CHECK_NOTA_NO_ESCALA)->EnableWindow(TRUE);
}

// DISEÑO DE ARMONIA
void CArmonizadorDlg::OnDesactivarArmonia()
{
//    GetDlgItem(IDC_BOX_ARMONIA)->EnableWindow(FALSE);
	GetDlgItem(IDC_TEXT_INTERVALO_N)->EnableWindow(FALSE);
	GetDlgItem(IDC_EDIT_INTERVALO_N)->EnableWindow(FALSE);
	GetDlgItem(IDC_TEXT_INTERVALOS)->EnableWindow(FALSE);
	GetDlgItem(IDC_TEXT_INTERVALO1)->EnableWindow(FALSE);
	GetDlgItem(IDC_EDIT_INTERVALO1)->EnableWindow(FALSE);
	GetDlgItem(IDC_TEXT_INTERVALO2)->EnableWindow(FALSE);
	GetDlgItem(IDC_EDIT_INTERVALO2)->EnableWindow(FALSE);
	GetDlgItem(IDC_TEXT_INTERVALO3)->EnableWindow(FALSE);
	GetDlgItem(IDC_EDIT_INTERVALO3)->EnableWindow(FALSE);
	GetDlgItem(IDC_BUTTON_ARMONIA)->EnableWindow(FALSE);
}

// ESCALAS:
void CArmonizadorDlg::OnDesactivarTonosMayor()
{
	GetDlgItem(IDC_TONO_RE_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosMenor()
{
	GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_RE_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosDorico()
{
	GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosFrigio()
{
	GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_RE_B)->EnableWindow(FALSE);
```

```
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosLidio()
{
	GetDlgItem(IDC_TONO_DO_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_RE_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosMixolidio()
{
	GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_RE_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_S)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnDesactivarTonosLocrio()
{
	GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_RE_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_MI_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(FALSE);
	GetDlgItem(IDC_TONO_LA_B)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnActivarTonosEscala()
{
	int escala = GetCheckedRadioButton(IDC_ESC_MAYOR,
					IDC_ESC_LOCRIO);
	switch(escala){
		case IDC_ESC_MAYOR:
			OnEscMayor();
			break;
		case IDC_ESC_MENOR_NATURAL:
			OnEscMenorNatural();
			break;
		case IDC_ESC_MENOR_ARMONICA:
			OnEscMenorArmonica();
			break;
		case IDC_ESC_MENOR_MELODICA:
			OnEscMenorMelodica();
			break;
		case IDC_ESC_JONICO:
			OnEscJonico();
			break;
		case IDC_ESC_DORICO:
			OnEscDorico();
			break;
		case IDC_ESC_FRIGIO:
			OnEscFrigio();
			break;
		case IDC_ESC_LIDIO:
			OnEscLidio();
			break;
```

```
            case IDC_ESC_MIXOLIDIO:
                  OnEscMixolidio();
                  break;
            case IDC_ESC_EOLICO:
                  OnEscEolico();
                  break;
            case IDC_ESC_LOCRIO:
                  OnEscLocrio();
                  break;
      }
}

// TONOS:
void CArmonizadorDlg::OnDesactivarTonos()
{
      GetDlgItem(IDC_BOX_TONO)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_DO_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_DO)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_DO_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_RE_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_RE)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_RE_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_MI_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_MI)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_MI_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_FA_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_FA)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_FA_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SOL)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SOL_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_LA_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_LA)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_LA_S)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SI_B)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SI)->EnableWindow(FALSE);
      GetDlgItem(IDC_TONO_SI_S)->EnableWindow(FALSE);
}
void CArmonizadorDlg::OnActivarTonos()
{
      GetDlgItem(IDC_BOX_TONO)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_DO_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_DO)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_DO_S)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_RE_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_RE)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_RE_S)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_MI_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_MI)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_MI_S)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_FA_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_FA)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_FA_S)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_SOL_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_SOL)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_SOL_S)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_LA_B)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_LA)->EnableWindow(TRUE);
      GetDlgItem(IDC_TONO_LA_S)->EnableWindow(TRUE);
```

```
	GetDlgItem(IDC_TONO_SI_B)->EnableWindow(TRUE);
	GetDlgItem(IDC_TONO_SI)->EnableWindow(TRUE);
	GetDlgItem(IDC_TONO_SI_S)->EnableWindow(TRUE);
}

void CArmonizadorDlg::OnIniciarTono()
{
	UpdateData(TRUE);
	m_tono = 0;
	UpdateData(FALSE);
	rtdxOutputData[2]=0;
}
void CArmonizadorDlg::OnActivarArmonia()
{
//	GetDlgItem(IDC_BOX_ARMONIA)->EnableWindow(TRUE);
	GetDlgItem(IDC_TEXT_INTERVALO_N)->EnableWindow(TRUE);
	GetDlgItem(IDC_EDIT_INTERVALO_N)->EnableWindow(TRUE);
	GetDlgItem(IDC_TEXT_INTERVALOS)->EnableWindow(TRUE);
	GetDlgItem(IDC_TEXT_INTERVALO1)->EnableWindow(TRUE);
	GetDlgItem(IDC_EDIT_INTERVALO1)->EnableWindow(TRUE);
	GetDlgItem(IDC_BUTTON_ARMONIA)->EnableWindow(TRUE);
}
void CArmonizadorDlg::OnActualizarArmonia()
{
	UpdateData(TRUE);
	switch(m_intervaloN){
		case 1:
			GetDlgItem(IDC_TEXT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_TEXT_INTERVALO2)->EnableWindow(FALSE);
			GetDlgItem(IDC_TEXT_INTERVALO3)->EnableWindow(FALSE);
			GetDlgItem(IDC_EDIT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO2)->EnableWindow(FALSE);
			GetDlgItem(IDC_EDIT_INTERVALO3)->EnableWindow(FALSE);
			break;
		case 2:
			GetDlgItem(IDC_TEXT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_TEXT_INTERVALO2)->EnableWindow(TRUE);
			GetDlgItem(IDC_TEXT_INTERVALO3)->EnableWindow(FALSE);
			GetDlgItem(IDC_EDIT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO2)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO3)->EnableWindow(FALSE);
			break;
		case 3:
			GetDlgItem(IDC_TEXT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_TEXT_INTERVALO2)->EnableWindow(TRUE);
			GetDlgItem(IDC_TEXT_INTERVALO3)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO1)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO2)->EnableWindow(TRUE);
			GetDlgItem(IDC_EDIT_INTERVALO3)->EnableWindow(TRUE);
			break;
		default:
			break;
	}
}
```

```
//*********************************************************************
//                            OPCIONES EFECTO
//*********************************************************************

void CArmonizadorDlg::OnEfBypass()
{
      OnDesactivarAcorde();
      OnDesactivarArmonia();
      rtdxOutputData[0]=0;
      OnEnviarDatos();
}

void CArmonizadorDlg::OnEfOctavador1()
{
      OnDesactivarAcorde();
      OnDesactivarArmonia();
      rtdxOutputData[0]=1;
      OnEnviarDatos();
}
void CArmonizadorDlg::OnEfOctavador2()
{
      OnDesactivarAcorde();
      OnDesactivarArmonia();
      rtdxOutputData[0]=2;
      OnEnviarDatos();
}
void CArmonizadorDlg::OnEfQuintas1()
{
      OnDesactivarAcorde();
      OnDesactivarArmonia();
      rtdxOutputData[0]=3;
      OnEnviarDatos();
}
void CArmonizadorDlg::OnEfQuintas2()
{
      OnDesactivarAcorde();
      OnDesactivarArmonia();
      rtdxOutputData[0]=4;
      OnEnviarDatos();
}
void CArmonizadorDlg::OnEfTriada()
{
      OnActivarAcorde();
      OnActivarTonosEscala();
      OnDesactivarArmonia();
      rtdxOutputData[0]=5;
      OnEnviarDatos();

}
void CArmonizadorDlg::OnEfCuatriada()
{
      OnActivarAcorde();
      OnActivarTonosEscala();
      OnDesactivarArmonia();
      rtdxOutputData[0]=6;
      OnEnviarDatos();
}
void CArmonizadorDlg::OnEfArmonia()
{
      OnActivarArmonia();
      OnActualizarArmonia();
```

```
	OnDesactivarAcorde();
	rtdxOutputData[0]=7;
	OnEnviarDatos();
}

//*********************************************************************
//                              OPCIONES ESCALA
//*********************************************************************

//ESCALAS:
void CArmonizadorDlg::OnEscMayor()
{
	OnActivarTonos();
	OnDesactivarTonosMayor();
	OnIniciarTono();
	rtdxOutputData[1]=0;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscMenorNatural()
{
	OnActivarTonos();
	OnDesactivarTonosMenor();
	OnIniciarTono();
	rtdxOutputData[1]=1;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscMenorArmonica()
{
	OnActivarTonos();
	OnDesactivarTonosMenor();
	OnIniciarTono();
	rtdxOutputData[1]=2;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscMenorMelodica()
{
	OnActivarTonos();
	OnDesactivarTonosMenor();
	OnIniciarTono();
	rtdxOutputData[1]=3;
	OnEnviarDatos();
}

//MODOS:
void CArmonizadorDlg::OnEscJonico()
{
	OnActivarTonos();
	OnDesactivarTonosMayor();
	OnIniciarTono();
	rtdxOutputData[1]=4;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscDorico()
{
	OnActivarTonos();
	OnDesactivarTonosDorico();
	OnIniciarTono();
	rtdxOutputData[1]=5;
	OnEnviarDatos();
}
```

```
void CArmonizadorDlg::OnEscFrigio()
{
	OnActivarTonos();
	OnDesactivarTonosFrigio();
	OnIniciarTono();
	rtdxOutputData[1]=6;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscLidio()
{
	OnActivarTonos();
	OnDesactivarTonosLidio();
	OnIniciarTono();
	rtdxOutputData[1]=7;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscMixolidio()
{
	OnActivarTonos();
	OnDesactivarTonosMixolidio();
	OnIniciarTono();
	rtdxOutputData[1]=8;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscEolico()
{
	OnActivarTonos();
	OnDesactivarTonosMenor();
	OnIniciarTono();
	rtdxOutputData[1]=9;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnEscLocrio()
{
	OnActivarTonos();
	OnDesactivarTonosLocrio();
	OnIniciarTono();
	rtdxOutputData[1]=10;
	OnEnviarDatos();
}

//*****************************************************************
//                          OPCIONES TONALIDAD
//*****************************************************************

void CArmonizadorDlg::OnTonoSiS()
{
	rtdxOutputData[2]=0;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoDo()
{
	rtdxOutputData[2]=0;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoDoS()
{
	rtdxOutputData[2]=1;
	OnEnviarDatos();
}
```

```
void CArmonizadorDlg::OnTonoReB()
{
	rtdxOutputData[2]=1;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoRe()
{
	rtdxOutputData[2]=2;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoReS()
{
	rtdxOutputData[2]=3;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoMiB()
{
	rtdxOutputData[2]=3;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoMi()
{
	rtdxOutputData[2]=4;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoFaB()
{
	rtdxOutputData[2]=4;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoMiS()
{
	rtdxOutputData[2]=5;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoFa()
{
	rtdxOutputData[2]=5;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoFaS()
{
	rtdxOutputData[2]=6;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoSolB()
{
	rtdxOutputData[2]=6;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoSol()
{
	rtdxOutputData[2]=7;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoSolS()
{
	rtdxOutputData[2]=8;
	OnEnviarDatos();
}
```

```
void CArmonizadorDlg::OnTonoLaB()
{
	rtdxOutputData[2]=8;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoLa()
{
	rtdxOutputData[2]=9;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoLaS()
{
	rtdxOutputData[2]=10;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoSiB()
{
	rtdxOutputData[2]=10;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoSi()
{
	rtdxOutputData[2]=11;
	OnEnviarDatos();
}
void CArmonizadorDlg::OnTonoDoB()
{
	rtdxOutputData[2]=11;
	OnEnviarDatos();
}

//*********************************************************************
//                     CHECK NOTA FUERA DE ESCALA
//*********************************************************************

void CArmonizadorDlg::OnCheckNotaNoEscala()
{
	UpdateData(TRUE);
	if (m_notaNoEscala==TRUE) {
		rtdxOutputData[3]=1; // Si esta activado suena la nota
	}
	if (m_notaNoEscala==FALSE) {
		rtdxOutputData[3]=0;// si no esta activado no suena la nota
	}
	OnEnviarDatos();
}

//*********************************************************************
//                                SLIDERS
//*********************************************************************

void CArmonizadorDlg::OnReleasedcaptureSliderGainIn(NMHDR* pNMHDR,
                                                    LRESULT* pResult)
{
	UpdateData(TRUE); //Actualiza todas las variables segun estado
	                  //de los componentes u objetos
	rtdxOutputData[4] = m_gainIn;
	OnEnviarDatos();
	*pResult = 0;
}
```

```
void CArmonizadorDlg::OnReleasedcaptureSliderVolumen(NMHDR* pNMHDR,
                                                     LRESULT* pResult)
{
      UpdateData(TRUE); //Actualiza todas las variables segun estado
                        //de los componentes u objetos
      rtdxOutputData[5] = m_volumen;
      OnEnviarDatos();
      *pResult = 0;
}

//*********************************************************************
//                           DISEÑO DE ARMONIA
//*********************************************************************

void CArmonizadorDlg::OnChangeEditIntervaloN()
{
      OnActualizarArmonia();
}
void CArmonizadorDlg::OnKillfocusEditIntervaloN()
{
      OnActualizarArmonia();
}
void CArmonizadorDlg::OnChangeEditIntervalo1()
{
      UpdateData(TRUE);
}
void CArmonizadorDlg::OnKillfocusEditIntervalo1()
{
      UpdateData(TRUE);
}
void CArmonizadorDlg::OnChangeEditIntervalo2()
{
      UpdateData(TRUE);
}
void CArmonizadorDlg::OnKillfocusEditIntervalo2()
{
      UpdateData(TRUE);
}
void CArmonizadorDlg::OnChangeEditIntervalo3()
{
      UpdateData(TRUE);
}
void CArmonizadorDlg::OnKillfocusEditIntervalo3()
{
      UpdateData(TRUE);
}

void CArmonizadorDlg::OnButtonArmonia()
{
      UpdateData(TRUE);
      rtdxOutputData[6] = m_intervaloN;
      rtdxOutputData[7] = m_intervalo1;
      rtdxOutputData[8] = m_intervalo2;
      rtdxOutputData[9] = m_intervalo3;
      OnEnviarDatos();
}
```

```
//*****************************************************************
//                          DETECTAR NOTA
//*****************************************************************

// Esta parte del programa aun no funciona correctamente
// Se deja para futuras mejoras


void CArmonizadorDlg::OnButtonDetNota()
{
/*
      OnRecibirDatos();
      CString strPos;
      strPos.Format ("%f", rtdxInputData[0]);
      MessageBox(strPos);
      CString strPos2;
      strPos2.Format ("%f", rtdxInputData[1]);
      MessageBox(strPos2);
*/

      if(a==0){
          rtdxOutputData[10] = 1;
          OnEnviarDatos();
          a=1;
      }else{

      CString notas [12] = {"Do","Do#","Re","Re#","Mi","Fa","Fa#",
                                    "Sol","Sol#","La","La#","Si"};

      CString octava [11] = {"0","1","2","3","4","5",
                             "6","7","8","9","10"};
      OnRecibirDatos();
      UpdateData(TRUE);
      m_textFrecuencia = rtdxInputData[0];
      m_textNota = notas[(int)rtdxInputData[1]]
                        + octava[(int)rtdxInputData[2]];
      UpdateData(FALSE);
      a=0;
      }
}
```

armonizadorDlg.h

```
// armonizadorDlg.h : header file
//

#if
!defined(AFX_ARMONIZADORDLG_H__EF73CDE2_5716_4104_B71E_71D5B910EE70__I
NCLUDED_)
#define
AFX_ARMONIZADORDLG_H__EF73CDE2_5716_4104_B71E_71D5B910EE70__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

//#include <windows.h>

#include "rtdxint.h"          //comunicación RTDX
#define LONG_RTDX 20          //Longitud de las arrays de comunicación

// CArmonizadorDlg dialog

class CArmonizadorDlg : public CDialog
{

// DECLARACION DE VARIABLES:

float rtdxInputData[LONG_RTDX];
int rtdxOutputData[LONG_RTDX];
int a;

// Construction
public:

      IRtdxExp* w_RTDX;
      IRtdxExp* r_RTDX;

      CArmonizadorDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
      //{{AFX_DATA(CArmonizadorDlg)
      enum { IDD = IDD_ARMONIZADOR_DIALOG };
      CButton     m_detectarNota;
      CButton     m_aceptarArmonia;
      int         m_efecto;
      int         m_escala;
      int         m_tono;
      int         m_volumen;
      int         m_gainIn;
      BOOL  m_notaNoEscala;
      int         m_intervaloN;
      int         m_intervalo1;
      int         m_intervalo2;
      int         m_intervalo3;
      float m_textFrecuencia;
      CString     m_textNota;
      //}}AFX_DATA

      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CArmonizadorDlg)
      protected:
```

```
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
								//support
	//}}AFX_VIRTUAL

// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	//{{AFX_MSG(CArmonizadorDlg)
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	afx_msg void OnEnviarDatos();
	afx_msg void OnRecibirDatos();
	afx_msg void OnDesactivarAcorde();
	afx_msg void OnDesactivarArmonia();
	afx_msg void OnDesactivarTonos();
	afx_msg void OnDesactivarTonosMayor();
	afx_msg void OnDesactivarTonosMenor();
	afx_msg void OnDesactivarTonosDorico();
	afx_msg void OnDesactivarTonosFrigio();
	afx_msg void OnDesactivarTonosLidio();
	afx_msg void OnDesactivarTonosMixolidio();
	afx_msg void OnDesactivarTonosLocrio();
	afx_msg void OnActivarAcorde();
	afx_msg void OnActivarArmonia();
	afx_msg void OnActualizarArmonia();
	afx_msg void OnActivarTonos();
	afx_msg void OnActivarTonosEscala();
	afx_msg void OnIniciarTono();
	afx_msg void OnCheckNotaNoEscala();
	afx_msg void OnEfBypass();
	afx_msg void OnEfOctavador1();
	afx_msg void OnEfOctavador2();
	afx_msg void OnEfQuintas1();
	afx_msg void OnEfQuintas2();
	afx_msg void OnEfTriada();
	afx_msg void OnEfCuatriada();
	afx_msg void OnEfArmonia();
	afx_msg void OnEscMayor();
	afx_msg void OnEscMenorNatural();
	afx_msg void OnEscMenorArmonica();
	afx_msg void OnEscMenorMelodica();
	afx_msg void OnEscJonico();
	afx_msg void OnEscDorico();
	afx_msg void OnEscFrigio();
	afx_msg void OnEscLidio();
	afx_msg void OnEscMixolidio();
	afx_msg void OnEscEolico();
	afx_msg void OnEscLocrio();
	afx_msg void OnReleasedcaptureSliderGainIn(NMHDR* pNMHDR,
		LRESULT* pResult);
	afx_msg void OnReleasedcaptureSliderVolumen(NMHDR* pNMHDR,
		LRESULT* pResult);
	afx_msg void OnTonoDo();
	afx_msg void OnTonoDoB();
	afx_msg void OnTonoDoS();
	afx_msg void OnTonoRe();
	afx_msg void OnTonoReB();
```

```
	afx_msg void OnTonoReS();
	afx_msg void OnTonoMi();
	afx_msg void OnTonoMiB();
	afx_msg void OnTonoMiS();
	afx_msg void OnTonoFa();
	afx_msg void OnTonoFaB();
	afx_msg void OnTonoFaS();
	afx_msg void OnTonoSol();
	afx_msg void OnTonoSolB();
	afx_msg void OnTonoSolS();
	afx_msg void OnTonoLa();
	afx_msg void OnTonoLaB();
	afx_msg void OnTonoLaS();
	afx_msg void OnTonoSi();
	afx_msg void OnTonoSiB();
	afx_msg void OnTonoSiS();
	afx_msg void OnChangeEditIntervaloN();
	afx_msg void OnKillfocusEditIntervaloN();
	afx_msg void OnChangeEditIntervalo1();
	afx_msg void OnKillfocusEditIntervalo1();
	afx_msg void OnChangeEditIntervalo2();
	afx_msg void OnKillfocusEditIntervalo2();
	afx_msg void OnChangeEditIntervalo3();
	afx_msg void OnKillfocusEditIntervalo3();
	afx_msg void OnButtonArmonia();
	afx_msg void OnButtonDetNota();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
//   Microsoft   Visual   C++   will   insert   additional   declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_ARMONIZADORDLG_H__EF73CDE2_5716_4104_B71E_71D5B910EE70__I
NCLUDED_)
```